National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service

Services des thèses canadiennes

Ottawa, Canada
K1A 0N4

CANADIAN THESES

THÈSES CANADIENNES

NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30.

AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30.

THIS DISSERTATION
HAS BEEN MICROFILMED
EXACTLY AS RECEIVED

LA THÈSE A ÉTÉ
MICROFILMÉE TELLE QUE
NOUS L'AVONS REÇUE

NL-339 (r.86/06)

Canada

THE UNIVERSITY OF ALBERTA

NEUTRAL OPENINGS:
A GENERAL THEORY

BY

TRENT KAISER

©

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND
RESEARCH IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF MASTER OF SCIENCE.

DEPARTMENT OF MECHANICAL ENGINEERING

EDMONTON, ALBERTA
SPRING 1987

Permission has been granted to the National Library of Canada to microfilm this thesis and to lend or sell copies of the film.

The author (copyright owner) has reserved other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without his/her written permission.

L'autorisation a été accordée à la Bibliothèque nationale du Canada de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur (titulaire du droit d'auteur) se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation écrite.

ISBN 0-315-37762-3

THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: TRENT KAISER

TITLE OF THESIS: NEUTRAL OPENINGS: A GENERAL THEORY

DEGREE: MASTER OF SCIENCE

YEAR THIS DEGREE GRANTED: 1987

Permission is hereby granted to THE UNIVERSITY OF ALBERTA LIBRARY to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

Trent Kaiser

....

4507 54 Avenue
Wetaskiwin, Alberta
T9A 0Y9

Date: January 9 1987

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled NEUTRAL OPENINGS: A GENERAL THEORY submitted by TRENT KAISER in partial fulfilment of the requirements for the degree of MASTER OF SCIENCE.

Date: Sept. 25. 1986

To my parents

Gordon and Mary "Sue" Kaiser.

ABSTRACT

Designers are often faced with the necessity of introducing openings into panels which may compromise their structural integrity. Efforts to reinforce the opening, though helpful, usually leave the stress field altered and some stress concentration remains.

This thesis describes a general theory for reinforcing openings in a special manner to leave the stress field undisturbed. These are referred to as neutral openings. An example of the theory is applied to a panel subject to simple tension. Several versions of this opening are analysed using the finite element method. The purpose of the finite element analysis is to assess the feasibility of practical applications of neutral openings, particularly with respect to considerations not dealt with in the general theory.

The finite element program used in the analysis was developed and run on a simple personal computer. The development of the program is also described.

ACKNOWLEDGEMENTS

The author wishes to thank his supervisor Dr. David Budney under whom the research for this thesis was carried out. His guidance and encouragement are deeply appreciated.

The author would also like to thank Darlene Hyrve for her assistance in typing the thesis.

TABLE OF CONTENTS

CHAPTER PAGE

I. NEUTRAL OPENING THEORY 1
- Introduction 1
- Reinforcement Loads 2
- Reinforcement Strains and Rotation ... 10
- Panel Strains 12
- Panel Rotation 13
- Reinforcement Characteristics 17
- Practical Considerations 19
- Example 22

II. NUMERICAL ANALYSIS OF A NEUTRAL HOLE 29
- Stress Function 29
- Opening Shape 30
- Stress Function Constants 32
- Reinforcement Cross Section 34
- Reinforcements Configurations 35
- Junction Configurations 39
- Simple Neutral Opening 42
- Numerical Analysis 43

III. FINEL - FINITE ELEMENT PROGRAM 51

Introduction 51

Overview 51

Element 52

Global Stiffness Matrix 54

System Solver 55

Stress Field Evaluation 55

Sample Application of Finel 57

IV FINITE ELEMENT ANALYSIS RESULTS 64

General 64

Unreinforced Circular Opening 67

Bulky Reinforcement with Integral Compression Member 70

Bulky Reinforcement with Compresson Member Insert 80

Compact Reinforcement with Integral Compression Member 86

Compact Reinforcement with Compresson Member Insert 93

Compact Reinforcement with Sliding Interface at Tension Junction 99

Simple Rectangular Reinforcing Liner .104

Summary 111

CONCLUSIONS 113

REFERENCES 115

APPENDIX A. 8 Node, Quadratic Isoparametric
Planes Stress or Strain Element 118
APPENDIX B. Global System Formation 131
APPENDIX C. Global System Solver 135
APPENDIX D. Program Listing for Finel 140

LIST OF TABLES

TABLE DESCRIPTION PAGE

4-1 Stress Concentration Summary 112

LIST OF FIGURES

FIGURE	TITLE	PAGE
1-1a	Panel Loaded in Plane	3
1-1b	Panel With Neutral Opening	3
1-2	Free Body Diagram of Reinforcement Element.	5
1-3	Rotation of Opening Boundary	14
1-4	Load on End of Reinforcement	21
1-5	Neutral Opening - Two Parabolas	24
1-6	Neutral Opening - Four Parabolas	26
1-7	One Quadrant of the Neutral Opening	27
2-1	Reinforcement Cross Section	36
2-2a	Bulky Channel Reinforcement Dimensions	37
2-2b	Compact Channel Reinforcement Dimensions ..	38
2-3	Tension and Compression Junctions	41
2-4a	F.E. Mesh - Bulky Liner	45
2-4b	F.E. Mesh - Panel with Bulky Liner	46
2-5a	F.E. Mesh - Compact Liner	47
2-5b	F.E. Mesh - Panel with Compact Liner	48
2-6a	F.E. Mesh - Simple Rectangular Liner	49
2-6b	F.E. Mesh - Panel with Simple Liner	50

3-1 F.E. Mesh - Panel with Circular Opening ... 59
3-2 Vertical Stresses - Analytical Solution ... 60
3-3 Vertical Stresses - Numerical Solution 61
3-4 Analytical Solution Near Opening 62
3-5 Numerical Solution Near Opening 63

4-1 Effective Stress Near Circular Opening 68
4-2 Displacement Field Near Circular Opening .. 69
4-3 B-I Panel Stress Field 74
4-4 Tension Junction Behaviour 75
4-5 B-I Panel Displacement Field 76
4-6 B-I Liner Stress Field 77
4-7 B-I Compression Junction Stress Field 78
4-8 B-I Tension Junction Stress Field 79
4-9 B-U Panel Stress Field 82
4-10 B-U Panel Displacement Field 83
4-11 B-U Liner Stress Field 84
4-12 B-U Compression Junction Stress Field 85
4-13 C-I Panel Stress Field 88
4-14 C-I Panel Displacement Field 89
4-15 C-I Liner Stress Field 90
4-16 C-I Compression Junction Stress Field 91
4-17 C-I Tension Junction Stress Field 92
4-18 C-U Panel Stress Field 95
4-19 C-U Panel Displacement Field.............. 96
4-20 C-U Liner Stress Field 97

4-21 C-U Compress Junction Stress Field 98

4-22 C-U-U Panel Stress Field 101

4-23 C-U-U Panel Displacement Field 102

4-24 C-U-U Tension Junction Stress Field 103

4-25 S-U Panel Stress Field 106

4-26 S-U Panel Displacement Field 107

4-27 S-U Liner Stress Field 108

4-28 S-U Compression Junction Stress Field 109

4-29 S-U Tension Junction Stress Field 110

NOMENCLATURE

$\sigma_x, \sigma_y, \sigma_{xy}$ - Panel stresses

Φ - Airy stress function

$f(x)$ - Opening shape function

t_p - Panel thickness

N - Reinforcement longitudinal load

V - Reinforcement shear load

M - Reinforcement bending load

s,t - Local coordinate system parallel and normal to opening boundary

θ - Angle between s and x axes

A - Cross-sectional area of reinforcement

I' - Bending stiffness of reinforcement

μ - Shear stiffness of reinforcement

ν - Poisson's ratio

E - Elastic modulus

ϵ_e - Extensional strain in reinforcement

ϕ_1 - Rate of rotation in reinforcement

γ_1 - Shear strain in reinforcement

ϵ_s - Extensional strain in panel on opening boundary

ϕ_{st} - Rotation of opening boundary

γ_{st} - Shear strain in panel on opening boundary

r - Radius of curvature of reinforcement

CHAPTER I

NEUTRAL OPENING THEORY

Introduction

Reduction of stress concentration factors caused by cutouts in tension panels has concerned designers for some time. Usually the approach consists of optimization of a reinforcement (liner) attached to the perimeter of the cutout. An annulus is the most common such reinforcement.

Neutral opening theory was first presented by Mansfield [1] in 1953. In this paper he described how certain reinforced holes could be made which would not alter the stress distribution in the main body of the sheet. His major simplifying assumption, that the reinforcement was compact, imposed restrictions on the opening shape and limited the practical applications of his theory.

Richards and Bjorkman [2] used the same approach as Mansfield. However by introducing the equations of Bresse [3], the reinforcement was modelled as a curved beam and the restrictions of the compact reinforcement assumption were overcome. Their paper described an approach for developing neutral hole theory but does not present any general expressions for neutral openings. Some of the equations used in their development are incorrect, notably their strain transformation and elastic rotation equations necessary for compatibility. The paper did not consider tension and compression elements which are necessary to provide loads at discontinuities in the reinforcement.

This chapter develops neutral hole theory further than was done by Mansfield and by Bjorkman and Richards. The result of the development is a set of formulae defining the cross-sectional characteristics of the reinforcement which are completely general in that they can be applied to any stress field which can be prescribed by the Airy stress function and for any shape of opening. Also presented are a new set of expressions for the tension and compression members necessary at reinforcement discontinuities which are also general and account for the flexural stiffness of the curved beam reinforcement model.

Reinforcement Loads

Consider a two dimensional panel of thickness t_p consisting of a homogeneous, isotropic material subject to loads in the plane (Fig. 1-1a). If the panel is thin it is in a state of plane stress and the stresses may be expressed as partial derivatives of the stress function, Φ:

$$\begin{aligned} \sigma_x &= \Phi_{yy} \\ \sigma_y &= \Phi_{xx} \\ \sigma_{xy} &= -\Phi_{xy} \end{aligned} \qquad (1)$$

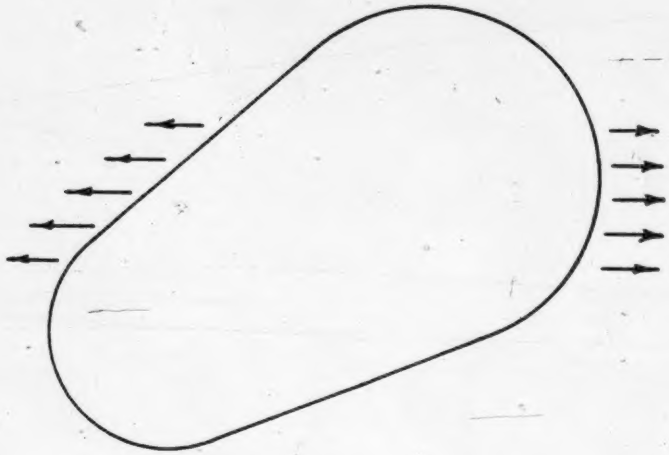


Fig. 1-1a
Panel Subject to In-Plane Loads

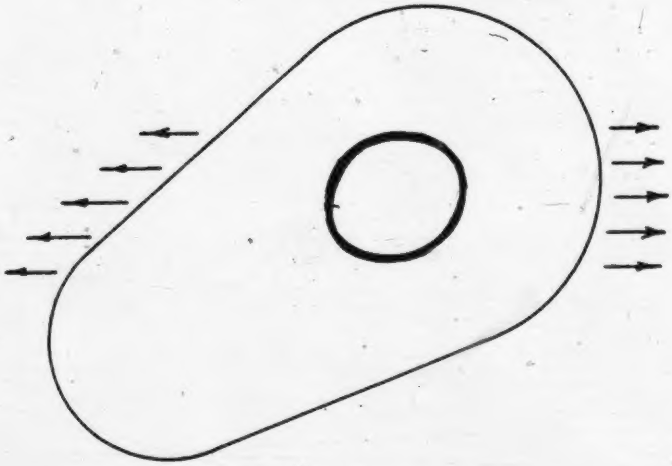


Fig. 1-1b
Panel with Opening Reinforced
to Leave Stress Field Unaltered

Now consider the same panel with a reinforced opening subject to an identical stress field (Fig. 1-1b). The shape of the opening is expressed using the shape function:

$$y = f(x) \tag{2}$$

which is assumed to be piecewise continuous and differentiable. An infinitesimal portion of the panel including the attached reinforcement is shown in Fig. 1-2. If it is assumed that the liner is chosen so that the stress field is undisturbed, then equilibrium of the elements in the x and y directions produces:

$$d(N\cos\theta) + d(V\sin\theta) = t_p(\sigma_{xy}dx - \sigma_x dy)$$
$$d(N\sin\theta) - d(V\cos\theta) = t_p(\sigma_y dx - \sigma_{xy}dy)$$

Substituting expressions (1) for the stress components:

$$d(N\cos\theta) + d(V\sin\theta) = -t_p(\Phi_{xy}dx + \Phi_{yy}dy) \tag{3}$$

$$d(N\sin\theta) - d(V\cos\theta) = t_p(\Phi_{xx}dx + \Phi_{xy}dy)$$

Since the right side of expressions 3 are total differentials in terms of the stress function, these can both be integrated directly giving:

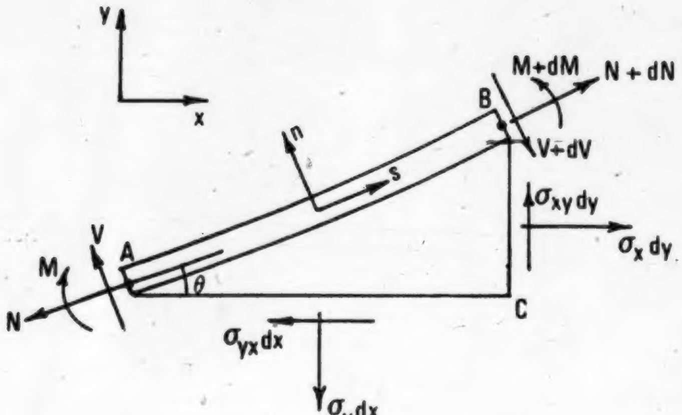

y
x
n
s
B
M+dM
N + dN
V+dV
$\sigma_{xy}\,dy$
$\sigma_x\,dy$
V
A
M
N
θ
C
$\sigma_{yx}\,dx$
$\sigma_y\,dx$

Fig. 1-2
Free Body Diagram of Reinforcement Element

$$N \frac{dx}{ds} + V \frac{dy}{ds} = -t_p(\Phi_y + b)$$
$$N \frac{dy}{ds} - V \frac{dx}{ds} = t_p(\Phi_x + a) \qquad (4)$$

the constants a and b being arbitrary constants of integration. Solving these two equations for the normal and shear forces yields:

$$\frac{N}{t_p} = (\Phi_x + a) \frac{dy}{ds} - (\Phi_y + b) \frac{dx}{ds}$$
$$\frac{V}{t_p} = -(\Phi_x + a) \frac{dx}{ds} - (\Phi_y + b) \frac{dy}{ds} \qquad (5)$$

The bending moment in the liner can be found by taking moments about point A in Fig. 1-2. Ignoring higher order terms:

$$dM + N\sin^{\theta} dx - N\cos^{\theta} dy - V\sin^{\theta} dy - V\cos^{\theta} dx = 0 \qquad (6)$$

Substituting (4) into (6) produces a differential form for M:

$$dM + t_p(\Phi_x + a) + t_p(\Phi_y + b)dy = 0$$

which can be integrated to give an expression for M:

$$\frac{M}{t_p} = -\Phi - ax - by - c \qquad (7)$$

Again, c is an arbitrary constant of integration. Notice that if M is differentiated with respect to s, the expression for V results, as expected.

Because the constants a, b, and c are arbitrary, and because the same effect of these constants on the liner cross-section can be made by arbitrary constants in the stress function, they can be set to zero without loss in generality of shape.

For the unit vector, t, normal to the opening shape and pointing from the panel into the opening:

$$\frac{dx}{dt} = \frac{dy}{ds}$$

$$\frac{dy}{dt} = \frac{-dx}{ds}$$

Using these identities, the expression for N can be written as:

$$\frac{N}{t_p} = \frac{d\Phi}{dt}$$

Similarly, the expression for V is:

$$\frac{V}{t_p} = \frac{-d\Phi}{ds}$$

These expressions, along with (7), illustrate the loads experienced by the reinforcement in terms of the shape function and stress function and are helpful in choosing the shape of the opening for a given stress field to optimize the liner cross-sectional characteristics.

The shape of the reinforcement is the same as that of the hole. Therefore dx/ds and dy/ds are expressed as functions of x using expression (2):

$$\frac{dx}{ds} = \frac{1}{(f'^2 + 1)^{1/2}} \tag{8}$$

$$\frac{dy}{ds} = \frac{f'}{(f'^2 + 1)^{1/2}}$$

Upon substitution of (8) into (5) and setting a, b, and c to zero, expressions for N, V, and M are found in terms of x and Φ:

$$\frac{N}{t_p} = \frac{(f'\Phi_x - \Phi_y)}{(f'^2 + 1)^{1/2}} \tag{9}$$

$$\frac{V}{t_p} = \frac{(\Phi_x + f'\Phi_y)}{(f'^2 + 1)^{1/2}} \tag{10}$$

$$\frac{M}{t_p} = -\Phi \tag{11}$$

Thus far, there have been no restrictions placed on the shape of the hole. Providing a reinforcing model can be found which can sustain the above loads, it is possible to reinforce an opening of arbitrary shape in a panel subject to a given stress field in such a manner so as not to cause the stress field to be disturbed by the opening.

At this point it is interesting to compare these results with those which Mansfield developed for a reinforcement with no flexural stiffness. Since no bending load may be carried by the compact liner:

$$M = 0$$

therefore:

$$\Phi = 0 \qquad (12)$$

and since $\frac{dM}{ds} = V$,

$$V = 0 \qquad (13)$$

Substituting into 10:

$$f' = \frac{-\Phi_x}{\Phi_y} \qquad (14)$$

Hence, if M is specified as in Mansfield's case where it must be zero, the shape of the opening is no longer arbitrary. Finally, the expression for N in such a liner is:

$$N = -(\Phi_x^2 + \Phi_y^2)^{1/2} \tag{15}$$

Equations (12) through (15) are the same as those developed by Mansfield, indicating that Mansfield's neutral hole is a special case of the neutral opening developed here.

Reinforcement Strains and Rotation

For compatibility it is necessary that the strains in the liner match those in the panel at the interface. In this analysis a curved beam of an elastic homogeneous isotropic material is chosen for the liner. The cross-sectional characteristics of the beam include:

a) A - The beam area which represents the longitudinal stiffness of the beam.

b) μ - The beam bulk shear factor, representing the shear stiffness of the beam defined from:

$$\gamma_{st} = \frac{V}{\mu AG}$$

c) I' - The bending stiffness of the curved beam is expressed using this factor which is formally defined as:

$$I' = \int_A \frac{y^2}{1 - \frac{y}{r}} \, dA$$

Here r is the radius of curvature at the centroid of the beam and y is the position with respect to the centerline measured toward the center of curvature. I' is approximately equal to the moment of inertia if the beam's radius of curvature is large in comparison with the thickness of the beam.

The deflections at the neutral axis of the beam are described by the generalized equations of Bresse [3]:

$$\varepsilon_{1s} = \frac{(N + \frac{M}{r})}{AE_1} \tag{16}$$

$$\frac{d\phi_1}{ds} = \frac{M}{EI'} + \frac{M}{EAr^2} + \frac{N}{AEr} \tag{17}$$

$$\gamma_{1s} = \frac{2(1 + \nu_1)V}{\mu AE} \tag{18}$$

The radius of curvature, r, can be expressed in terms of the shape function of the opening:

$$r = \frac{(f'^2 + 1)^{3/2}}{f''}$$

Substituting (9), (10) and (11) into (16), (17) and (18) yields expressions for the liner strains in terms of the stress function and shape functions only:

$$\varepsilon_1 = \frac{t_p[(f'^2 + 1)(f'\Phi_x - \Phi_y) - f''\Phi]}{AE_1(f'^2 + 1)^{3/2}} \qquad (19)$$

$$\frac{d\phi_1}{ds} = \frac{-t_p\Phi}{E_1 I'} + \frac{t_p f''[(f'^2 + 1)(f'\Phi_x - \Phi_y) - f''\Phi]}{E_1 A(f'^2 + 1)^3} \qquad (20)$$

$$\gamma_1 = \frac{-t_p(1 + \nu_1)(\Phi_x + f'\Phi_y)}{\mu AE_1(f'^2 + 1)^{1/2}} \qquad (21)$$

Panel Strains

The panel strains at the interface are found using the standard transformation equations:

$$\varepsilon_s = \frac{(1 - \nu_p)}{2E_p}(\Phi_{xx} + \Phi_{yy}) - \frac{(1 + \nu_p)}{2E_p}(\Phi_{xx} - \Phi_{yy})\cos 2\theta \qquad (22)$$

$$- \frac{(1 + \nu_p)}{E_p}\Phi_{xy}\sin 2\theta$$

$$\gamma_{st} = \frac{2(1 + \nu_p)}{E_p}(\Phi_{xx} - \Phi_{yy})\sin 2\theta - \frac{2(1 + \nu_p)}{E_p}\Phi_{xy}\cos 2\theta \quad (23)$$

Collecting common derivative terms and simplifying:

$$\varepsilon_s = \frac{[(f'^2 - \nu_p)\Phi_{xx} + (1 - \nu_p f'^2)\Phi_{yy} - 2(1 + \nu_p)f'\Phi_{xy}]}{E_p(f'^2 + 1)} \quad (24)$$

$$\gamma_{st} = \frac{2(1 + \nu_p)[f'(\Phi_{xx} - \Phi_{yy}) + (f'^2 - 1)\Phi_{xy}]}{E_p(f'^2 + 1)} \quad (25)$$

Panel Rotation

For compatibility of rotations in the liner and panel, an expression for the rotation of the panel at the opening boundary is required. Consider a small line element, ds, on the opening boundary of the panel (Fig. 1-3). When the panel is loaded, the boundary element moves from AB to A'B' with displacements, u and v, in terms of the s-t coordinate system. The rotation of the element is:

$$\phi = \frac{\partial v}{\partial s}$$

$$\phi = \frac{1}{2}\left(\frac{\partial v}{\partial s} - \frac{\partial u}{\partial t}\right) + \frac{1}{2}\left(\frac{\partial v}{\partial s} + \frac{\partial u}{\partial t}\right)$$

$$\phi = -\omega_{st} + \frac{1}{2}\gamma_{st}$$

The rate of change of rotation of the opening boundary is then:

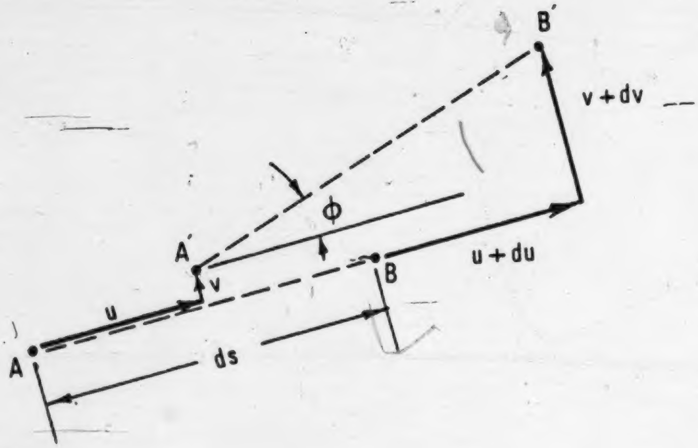

B'
v + dv
ϕ
A'
v
B
u + du
u
A
ds

Fig. -1-3
Rotation of Opening Boundary

$$\frac{d\phi}{ds} = \frac{-d\omega_{st}}{ds} + \frac{1}{2}\frac{d\gamma_{st}}{ds} \tag{26}$$

Because the material rotation is invariant,

$$\omega_{st} = \omega_{xy}$$

then

$$\frac{d\omega_{st}}{ds} = \frac{\partial\omega_{xy}}{\partial x}\frac{dx}{ds} + \frac{\partial\omega_{xy}}{\partial y}\frac{dy}{ds} \tag{27}$$

Expressing the material strains in terms of the panel stresses, and hence the stress function:

$$\gamma_{xy} = \frac{\partial v}{\partial x} + \frac{\partial u}{\partial y} = \frac{-2(1 + \nu)}{E}\frac{\partial^2\Phi}{\partial x\partial y} \tag{28}$$

$$\varepsilon_x = \frac{\partial u}{\partial x} = \frac{1}{E}\quad\frac{\partial^2\Phi}{\partial y^2} - \nu\,\frac{\partial^2\Phi}{\partial x^2} \tag{29}$$

$$\varepsilon_y = \frac{\partial v}{\partial y} = \frac{1}{E}\quad\frac{\partial^2\Phi}{\partial x^2} - \nu\,\frac{\partial^2\Phi}{\partial y^2} \tag{30}$$

The partial derivatives of the material rotation can be expressed in terms of the derivatives of the displacements; for example:

$$-\frac{\partial\omega_{xy}}{\partial x} = \frac{1}{2}\,\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 u}{\partial x\partial y} - 2\,\frac{\partial^2 u}{\partial x\partial y} \tag{31}$$

Substituting equations (28) through (30) into (31) produces the derivative of the boundary rotation with respect to x in terms of the stress function:

$$\frac{\partial\omega}{\partial x} = \frac{-1}{2E}\left\{-2(1+\nu)\frac{\partial^3\Phi}{\partial x^2\partial y} - \frac{\partial^3\Phi}{\partial y^3} + 2\nu\frac{\partial^3\Phi}{\partial x^2\partial y}\right\}$$

$$\frac{\partial\omega}{\partial x} = \frac{-1}{E}\frac{\partial}{\partial y}(\nabla^2\Phi)$$

And in a similar fashion it can be shown that:

$$\frac{\partial\omega}{\partial y} = \frac{1}{E}\frac{\partial}{\partial x}(\nabla^2\Phi)$$

Finally, substituting (25) for the panel shear strain and (27) into (26) produces the expression for the panel rotation at the opening boundary in terms of the stress function and opening shape function only:

$$\frac{d\gamma_{st}}{ds} = \frac{2(1+\nu_p)}{E_p(f'^2+1)^{5/2}}\left\{\begin{array}{l} f''[(1-f'^2)(\Phi_{xx}-\Phi_{yy}) + 4f'f''\Phi_{xy}] \\ + (f'^2+1)\left\{\begin{array}{l} f'(\Phi_{xxx} - f'\Phi_{yyy}) \\ + (2f'^2-1)\Phi_{xxy} \\ + f'(f'^2-2)\Phi_{xyy} \end{array}\right\} \end{array}\right\} \qquad (32)$$

$$\frac{d\omega_{st}}{ds} = \frac{f'\Phi_{xxx} - \Phi_{xxy} + f'\Phi_{xyy} - \Phi_{yyy}}{E_p(f'^2+1)^{1/2}}$$

$$\frac{d\phi}{ds} = \frac{m(f,\Phi)}{E_p(f'^2+1)^{5/2}} \qquad (33)$$

Where:

$$
\begin{aligned}
m(f,\Phi) = \{(1+\nu_p)\, f''[(1-f'^2)(\Phi_{xx}-\Phi_{yy}) + 4f'\Phi_{xy}] \\
+ (f'^2+1)\big(f'\Phi_{xxx}[f'^2+(2+\nu_p)] \\
- \Phi_{yyy}[(2+\nu_p)f'^2+1] \\
+ \Phi_{xxy}[(1+2\nu_p)f'^2+(2+\nu_p)] \\
+ \Phi_{xyy}[(2+\nu_p)f'^2-(1+2\nu_p)]\big)\}
\end{aligned}
$$

Reinforcement Characteristics

The strains at the interface in the panel and liner neutral axis have been found and, for compatibility, must be equal. This produces three equations with the three liner properties, A, I' and μ, as the unknowns. Combining (19) and (24) and solving for A produces:

$$
A = \frac{t_p E_p[(f'^2+1)(f'\Phi_x - \Phi_y)f''\Phi]}{E_e(f'^2+1)^{1/2}\left\{\begin{matrix}\Phi_{xx}(f'^2-\nu_p)+\Phi_{yy}(1-\nu_p f'^2) \\ -\,2\Phi_{xy}f'(1+\nu_p)\end{matrix}\right\}} \tag{34}
$$

Relating the shear strain in the beam to the shear strain in the panel, and taking sign of shear deformation into account:

$$\gamma_{st} = -\gamma_1 \tag{35}$$

Substituting (21) and (25) into (35), and substituting (34) for A, an expression for the beam shear factor is obtained:

$$\mu = \frac{(f'^2 + 1)(\Phi_x + f'\Phi_y)\left\{\begin{matrix}\Phi_{xx}(f'^2 - \nu_p) + \Phi_{yy}(1 - \nu_p f'^2)\\ -2\,(1 + \nu_p)f'\Phi_{xy}\end{matrix}\right\}}{\dfrac{2(1 + \nu_p)}{(1 + \nu_1)}\left[\begin{matrix}[f'(\Phi_{xx} - \Phi_{yy}) + (f'^2 - 1)\Phi_{xy}]\\ \mathrm{X}\ [(f'^2 + 1)(f'\Phi_x - \Phi_y) - f'\Phi]\end{matrix}\right]} \tag{36}$$

Using (20) and (33), and again substituting (34) for A, the moment of inertia for the liner is found to be:

$$I' = \frac{t_p \dfrac{E_p}{E_1}(f'^2 + 1)^{5/2}\Phi}{\left\{\begin{matrix} f''\left\{\begin{matrix}\Phi_{xx}[f'^2(2 + \nu_p) - (1 + 2\nu_p)]\\ -\Phi_{yy}[f'^2(1 + 2\nu_p) - (2 + \nu_p)]\\ -6f'(1 + \nu_p)\Phi_{xy}\end{matrix}\right\} \\ - (f'^2 + 1)\left\{\begin{matrix}f'\Phi_{xxx}[f'^2 + (2 + \nu_p)]\\ -\Phi_{yyy}[(2 + \nu_p)f'^2 + 1]\\ +\Phi_{xxy}[(1 + 2\nu_p)f'^2 - (2 + \nu_p)]\\ +f'\Phi_{xyy}[(2 + \nu_p)f'^2 - (1 + 2\nu_p)]\end{matrix}\right\}\end{matrix}\right\}} \tag{37}$$

Because Mansfield used the simplifying assumption that the reinforcement was compact, the only cross-sectional characteristic determined by him was the area of the liner. He showed that for the compact liner, the shape of the opening is no longer arbitrary but must satisfy the condition stated in (14). Substituting this into (34) results in the same expression for the area as Mansfield's, demonstrating that his neutral hole is a special case of the development in this thesis.

Practical Considerations

While equations (34), (36) and (37) are sufficient to completely describe the liner characteristics required for the neutral hole, they apply to a specific stress function for a given opening shape and are not appropriate for any other stress field. It should also be noted that these expressions may not always produce practical results. For example, a negative reinforcement area may be specified by (34) in some situations. Clearly, this cannot be produced in reality. Other impossible characteristics sometimes result from (36) or (37) or, in some cases, a reinforcement which fills the entire opening is prescribed. Even when the dimensions of the prescribed reinforcement are not extreme, the designer's judgement in the selection of both a suitable shape for the opening and appropriate constants in the stress function to produce reasonable values for I', A and μ is important.

To overcome these problems, the designer might consider using only a section of [illegible] opening described by f(x). Fig. 1-4 shows one end of [illegible] a section with the reinforcement loads needed to maintain the stress field. The shear and normal forces can be replaced by vertical and horizontal forces, F_x and F_y:

$$F_x = N\sin\theta + V\cos\theta$$

$$F_y = N\cos\theta - V\sin\theta$$

Substituting (6a) and (6b) for N and V and replacing $\sin\theta$ and $\cos\theta$ with $\frac{dx}{ds}$ and $\frac{dy}{ds}$ in (6) and (6a), the forces become:

$$F_x = -t_p\Phi_y \qquad (38)$$

$$F_y = t_p\Phi_x$$

These loads could be applied by horizontal and vertical members which experience the same strains as the panel:

$$\varepsilon_{px} = \frac{(\Phi_{yy} - \nu\Phi_{xx})}{E_p}$$

$$\varepsilon_{py} = \frac{(\Phi_{xx} - \nu\Phi_{yy})}{E_p}$$

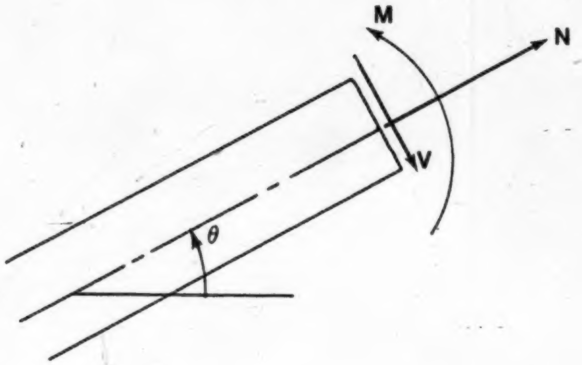


Fig. 1-4
Load State at End of Reinforcement

The load in the reinforcement is directly related to its area and the strain it encounters:

$$F_x = E_h A_h \varepsilon_x$$

$$F_y = E_v A_v \varepsilon_y \tag{39}$$

Replacing F_x and F_y in (39) with (38) and solving for the areas:

$$A_h = \frac{\frac{E_p}{E_h} t_p \Phi_y}{(\Phi_{yy} - \nu\Phi_{xx})}$$

$$A_v = \frac{\frac{E_p}{E_h} t_p \Phi_x}{(\Phi_{xx} - \nu\Phi_{yy})} \tag{40}$$

Example

Consider a panel subject to uniaxial tension. The stress function used is:

$$\Phi = \frac{\sigma}{2} (x^2 - ry)$$

The reinforcement used in the example is compact (the same as that which Mansfield used), so the shape of the opening is prescribed to be:

$$y = \frac{x^2}{r}$$

Obviously this opening shape is not closed, so sections must be used to close the hole. It may seem possible to accomplish this using only two parabolic sections, one the mirror image of the other with a compression element across the opening to provide the unbalanced horizontal force required at the discontinuity in the opening (Fig. 1-5). However, when (34) is used to calculate the cross-sectional area of the liner, it is found to be negative in the range:

$$-\frac{r}{2}\sqrt{\nu} < x_p < \frac{r}{2}\sqrt{\nu}_p$$

and undefined at

$$x = \pm \frac{r}{2}\sqrt{\nu}_p$$

Physically this means that the liner is expected to experience a tensile load while undergoing negative or zero extension in this region.

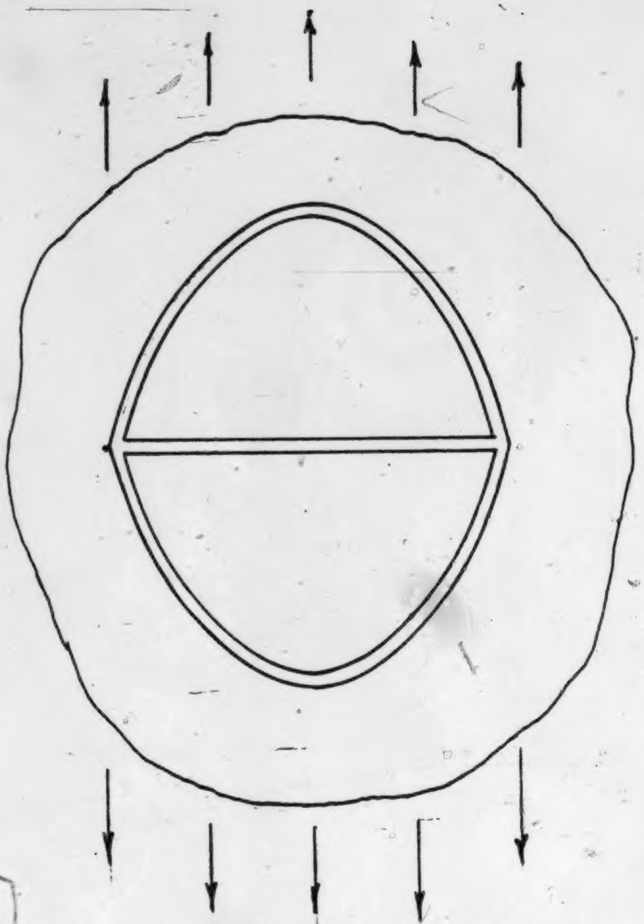


Fig. 1-5
Neutral Opening Consisting of
Two Parabolic Sections

The next possibility is to use four parabolic sections as shown in Fig. 1-6. A vertical tension element is required at the top and bottom to provide the unbalanced vertical force and, as in the first prototype, a compression element is required across the opening to provide the unbalanced horizontal force.

To further develop this opening, consider only one quadrant of the hole (Fig. 1-7). The section of the opening starts at the location of the tension element, x_t, and ends at the location of the compression member, x_c. It was shown from the first example that x_t must be greater than $(r/2)\sqrt{\nu_p}$ and, subject to this last constraint, the characteristics of the opening can be completely defined. From (34), the area of the liner is:

$$A_l = \frac{t_p E_p (4x^2 + r^2)^{3/2}}{2E_l (4x^2 - \nu_r{}^2)}$$

and using (40), the tension and compression member areas are:

$$A_T = 2A_V = \frac{2E_p t_p x_T}{E_T}$$

$$A_L = 2A_H = \frac{2E_p t_p \nu}{E_L \nu}$$

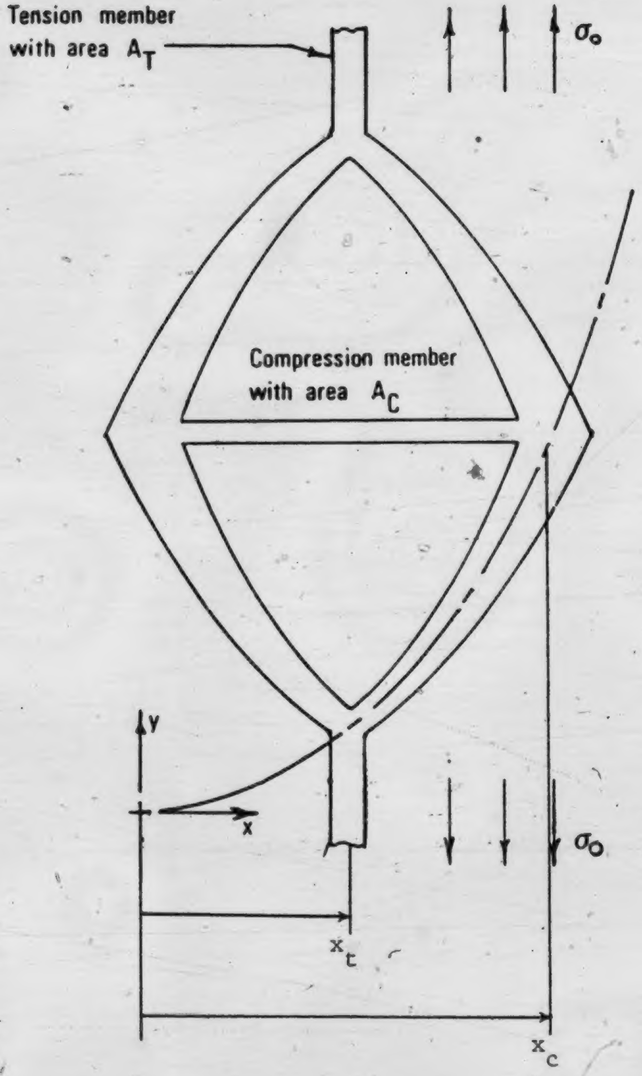

Tension member
with area A_T
σ_o
Compression member
with area A_C
y
x
σ_o
x_t
x_c

Fig. 1-6
Neutral Opening Consisting of
Four Parabolic Sections

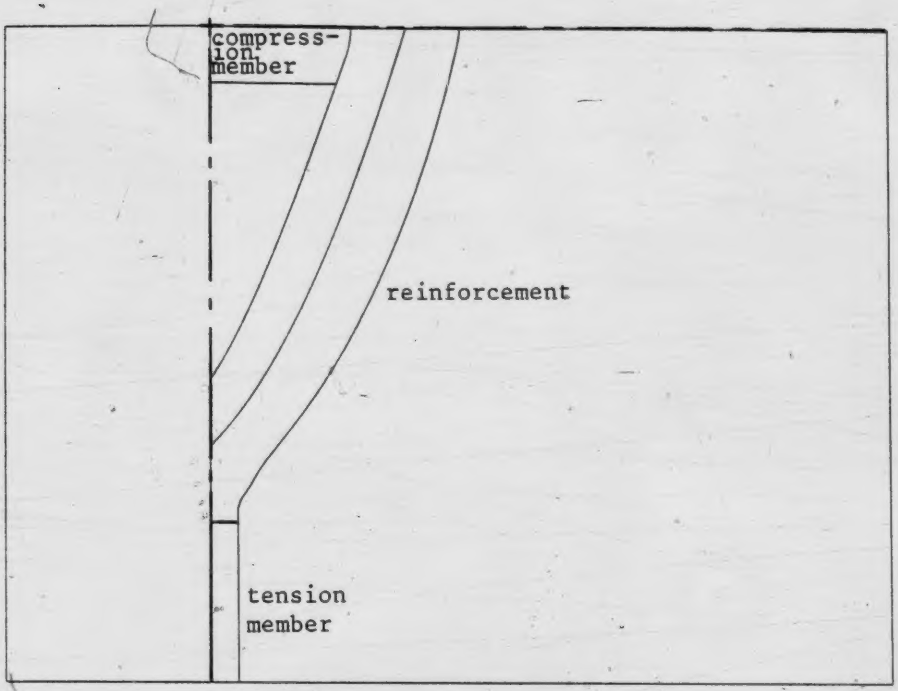

compress-
ion
member
reinforcement
tension
member

Fig. 1-7
One Quadrant of Neutral Opening

This example demonstrates that a practical neutral opening is indeed possible. Experiments using a reinforcement much stiffer than the panel to approximate a compact reinforcement has demonstrated that the theory is valid for the panel [4].

CHAPTER II

NUMERICAL ANALYSIS OF A NEUTRAL HOLE

A panel subject to uniform, uniaxial tension is a common component of many structures. The structural integrity of such panels is reduced by the introduction of an opening which may be necessary for a number of reasons. However, if neutral opening theory is applied to the design of the opening it may be possible to leave the structural integrity of the panel uncompromised.

This chapter details such an application of neutral opening theory and describes the numerical analysis performed to evaluate the application.

Stress Function

Given a panel located in the x-y plane and loaded in the y direction with a uniform stress σ, the complete stress function describing the stress field is:

$$\Phi = \frac{\sigma}{2} (x^2 + ax + by + c)$$

where a, b, and c are arbitrary constants.

Because the multiplying factor $\sigma/2$ is a common factor in the numerator and denominator of each of the expressions developed in the previous chapter for the reinforcement cross-sectional characteristics, it can be dropped from the function. Although valid for a specific opening shape and stress field, the liner characteristics produced by the neutral opening equations are not dependent upon the magnitude of the stress which the panel is subject to.

The arbitrary constants are retained to provide a means of adjusting the cross-section of the reinforcement. The design stress function is then:

$$\Phi = x^2 + ax \pm by + c \qquad (1)$$

Opening Shape

Several shape functions were considered for the opening. The first choice was the most common, a circular opening. However, for any constants chosen for the above stress function, the difficulties demonstrated in Chapter I with a parabolic opening were encountered, that is, positive loads are specified to be carried by the liner while accommodating negative strains. As a result, reinforcement characteristics were undefined in some regions of the arc of the opening and unreasonable in others. Other closed continuous shapes such as elliptical openings were unable to overcome this problem.

The final shape chosen was a parabolic shape similar to the Mansfield neutral opening. By visualizing the stress function as a contour map it is easier to understand the procedure to select the constants in the stress function. This particular stress function would then be a series of parabolas, which is why a parabolic shape is chosen. The shear and tensile loads in the reinforcement depend on the gradient of the stress field tangential and normal to the opening shape respectively and the bending moment is equal to the magnitude of the stress function. Thus, if the opening is located on a stress function contour other than that associated with the zero contour, a constant bending moment is applied to the reinforcement and reasonable cross sectional characteristics are specified.

However, since the opening shape has a continuously varying slope, the strains in the opening local coordinate system also vary and the cross sectional characteristics do not remain the same. Two approaches can be used to improve the liner to account for this. The opening can be moved in the stress field or the stress field can be moved with respect to the opening. It was decided to adjust the constants in the stress function to move the stress function contours slightly on the opening to produce a better cross section. As a result, the major load in the reinforcement is the longitudinal load with a small shear load and bending moment included.

The simplest parabolic opening shape function that can be used is:

$$y = \frac{x^2}{r}$$

where r, a constant shape parameter, was chosen to be 30 mm. in case a photoelastic model might be constructed.

Stress Function Constants

The selection of the arbitrary constants in the stress function involved a trial and error process. Using the expressions for the reinforcement loads the constants are modified and the cross section specification of the liner are updated until a reasonable set of dimensions are achieved over a suitable range.

Recalling the simple expressions for the reinforcement loads:

$$\frac{N}{t_p} = \frac{d\Phi}{dt}$$

$$\frac{V}{t_p} = \frac{-d\Phi}{ds}$$

$$\frac{M}{t_p} = -\Phi$$

If one considers a contour map of the stress function Φ, these expressions mean that the axial liner load decreases with the angle at which the opening cuts a contour line, the shear load increases with the angle at which the liner cuts the contour, and the bending moment in the liner is equal to the value of the contour line where the opening cuts the contour.

First consider a constant moment reinforcement. That is, the opening lies on a contour of the stress function. The bending moment is a constant, the shear load is zero and the bending moment is the gradient of Φ normal to the contour. Since,

$$f(x) = \frac{x^2}{r}$$

$$M = x^2 - ry + c$$

where c is the constant moment applied. This constant moment can be adjusted to specify a relatively compact reinforcement or a bulky reinforcement.

Further adjustments can be made to the liner by slightly adjusting r and by including a linear term in x. This causes the stress function contours to cross the opening boundary and results in a shear load and varying bending moment in the liner. This is desirable, since the rotation rate in the liner is not constant and some shear deformation will be present in the liner requiring ~~some~~ shear load. As a result, the region suitable for reinforcement can be greatly extended.

Reinforcement Cross-Section

To allow for comparison with future experimental analysis, material properties of a photoelastic material, $E = 2390$ MPa and $\nu = 0.39$, and a thickness of 6.7 mm. were used for the panel. The reinforcement was modelled using the same material properties as the panel.

To satisfy the equations of Bresse[3], the panel must be attached to the reinforcement at the centroid of its cross-section. There must also be enough variable parameters defining the cross-section to produce A, I', and μ specified by the reinforcement expressions.

The shape chosen for the cross-section is a channel shape shown in Fig. 2-1. The variable parameters allow specification of A and I', but not the shear stiffness parameter μ. This is not a serious problem since the dominant loads are tension and bending if the stress field constants are properly selected.

Reinforcements Configurations

Two sets of constants were used in the stress function to produce reinforcements with different characteristics. The first reinforcement was relatively square and bulky (reinforcement B). The other was a narrow more compact reinforcement, carrying a smaller bending load (reinforcement C). Fig. 2-2 shows the dimensions of the reinforcements with drawings of the cross section at representative locations, indicating the differences between the two models.

It is apparent that the reinforcement dimensions

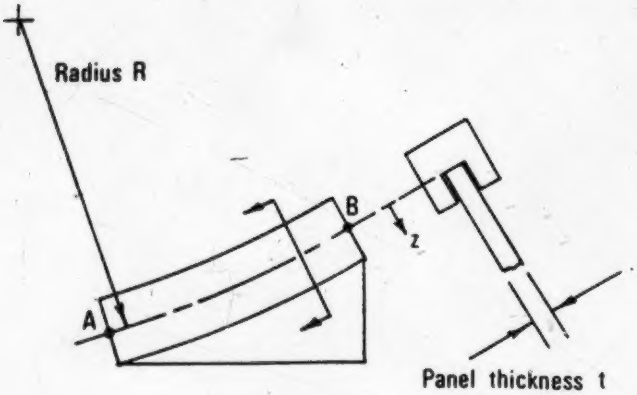
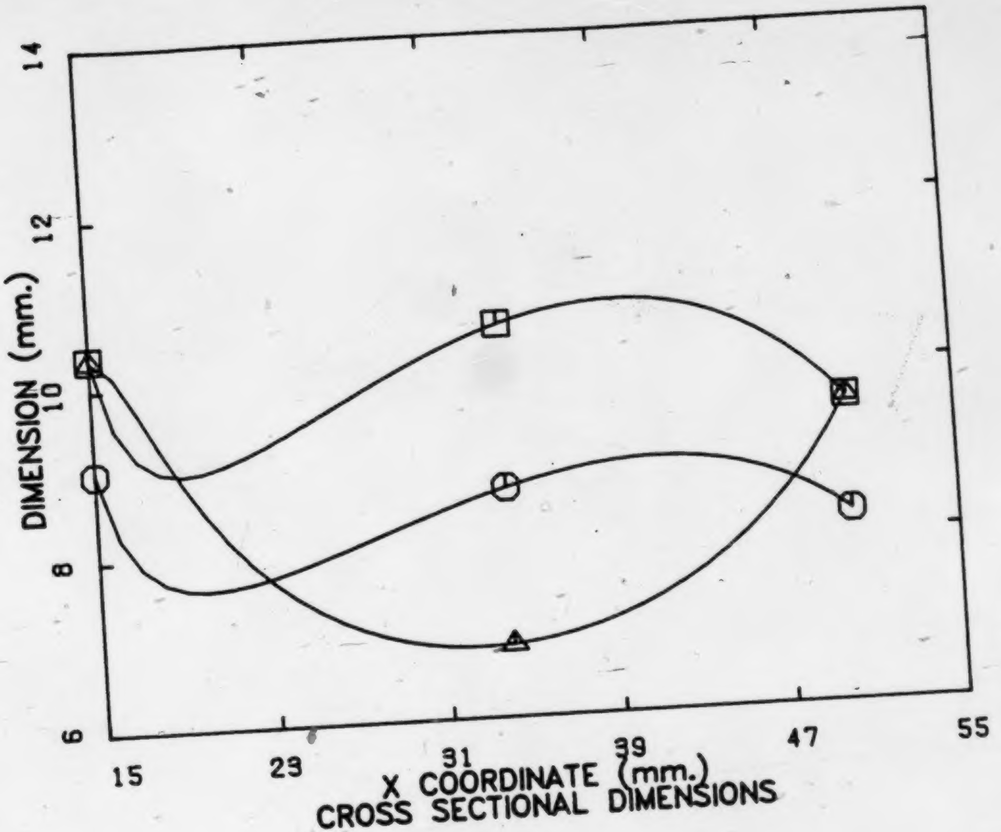

Radius R
B
z
A
Panel thickness t

Figure 2-1
Reinforcement Cross Section

The channel shape cross section provides enough variable parameters to specify the area and moment of inertia and still be attached at the neutral axis.



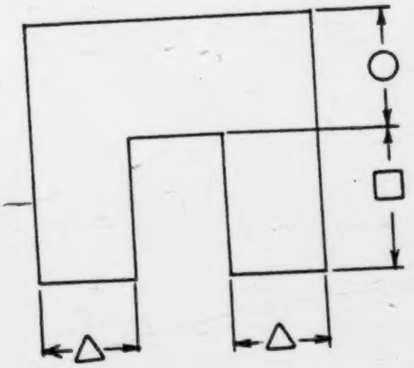


Figure 2-2-a
Bulky Channel Reinforcement
Cross Sectional Dimensions

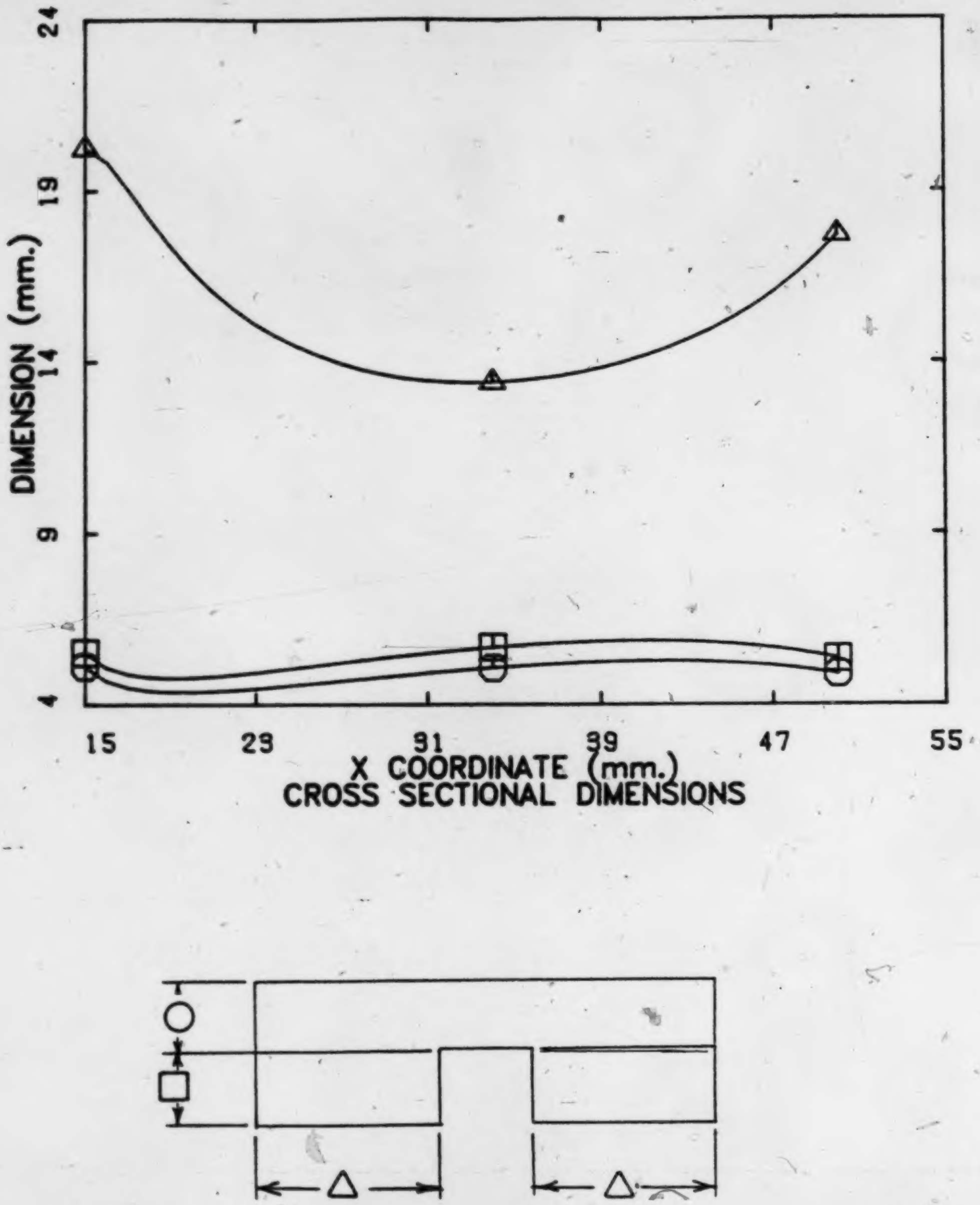


Figure 2-2-b
Compact Channel Reinforcement
Cross Sectional Dimensions

are defined only for a small range of x. The section of the parabola used is in the range $15 < x < 50$ and by using four symmetric sections with tension and compression members to provide equilibrating internal forces, a closed shape is produced as shown in Fig. 1-6.

Junction Configurations

The proportions of the reinforcement are specified by the neutral opening equations—only along the curve described by the shape function. These equations do not prescribe the shape of the junctions and, in particular, do not provide any direction to the designer as to how to deal with the joining of bulky members. Ideally the reinforcement would have no physical dimension, but instead would have the prescribed tensile, flexural and shear stiffness contained in the line of the opening shape. The junctions would simply be spring loaded hinges transferring equilibrating loads without being affected by the physical bulkiness of the members. Unfortunately, neither is possible.

To attach the reinforcing members in order to best resemble the theory, the vertical line of symmetry was placed where the opening shape begins (at $x = 15$) and the horizontal symmetry line was placed at the point where the opening shape function ends (at y = 83.333). The compressive member was placed inside the opening and the tension member was attached to the top of the panel. The tension and compression members were the same depth as the reinforcement at their respective junctions and their widths were determined by the specified area.

Two different types of junctions were compared to evaluate the performance of different junction configurations. One model considered the reinforcement and compression member as a one piece component. This integral configuration will be referred to as an I junction. The other used a sliding frictionless interface between the compression member and liner so only a normal force can be transferred. Such a junction is designated a U configuration since the components are unattached. This U junction was used at the tension junction for one neutral opening model as an interesting exercise, even though a frictionless tensile interface cannot be achieved in practice. Fig. 2-3 illustrates the variations in the model at the reinforcement junctions that were analysed.

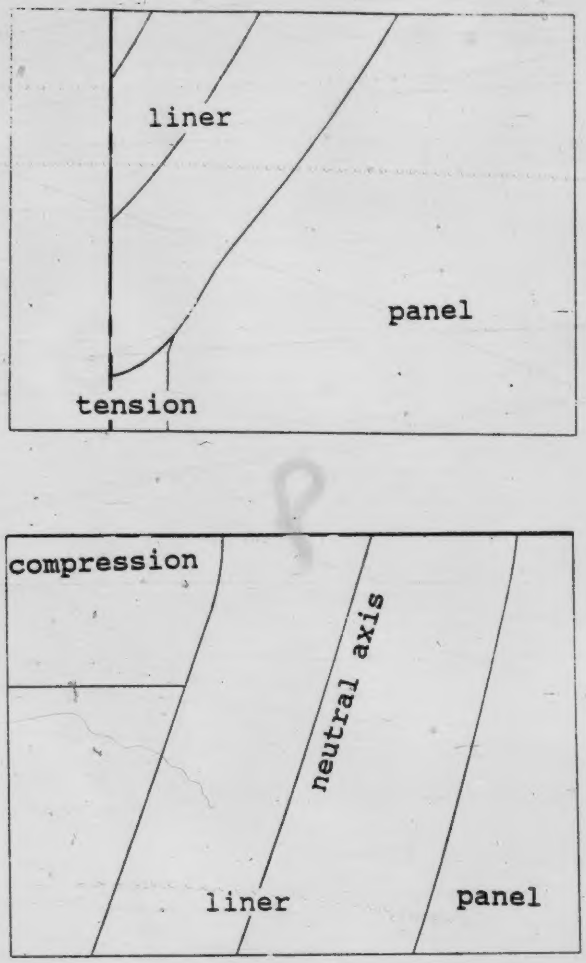

liner
panel
tension
compression
neutral axis
liner
panel

Figure 2-3
Tension and Compression Junctions

Simple Neutral Opening

Manufacture and assembly of the channel shape reinforcements would be complex and tedious. A simpler reinforcement to construct would have a rectangular cross section. This shape can also be adjusted to provide the area and moment of inertia specified by the neutral opening expressions, but does not satisfy the requirement that the liner be attached to the panel at its centroid. However, for the shape function and type of stress function employed, the dominant liner load is tension, hence the strain at the extreme axis of the reinforcement is approximately equal to that at its neutral axis.

This simplified reinforcement (S) was also modelled by the finite element method. The liner was attached to the opening boundary in such a way that all the liner material intruded into the opening. To compensate for this, the tension element was placed between the panel and the vertical line of symmetry to widen the opening. The tension element was then attached to a rigid member joining the symmetric reinforcing sections. This configuration is very desirable because by selecting an appropriate width for the tension element, its depth can be made equal to the panel thickness. In this way the reinforcing member appears as part of the panel. Its disadvantage is that it is a less accurate approximation of the ideal dimensionless junction.

The stress function was selected to produce a relatively compact reinforcement (depth : width = 3), and a compression member insert was used (sliding frictionless interface).

Numerical Analysis

The finite element program Finel, described in Chapter 3, was developed to analyse the stress field in the panel and reinforcement of the neutral opening assemblies. The size of the finite element model was reduced by utilizing the symmetry of the problem and modelling one quadrant of the opening.

Figs. 2-4 and 2-5 show the finite element meshes used in the analysis of the U-shape reinforcement neutral opening. The models use 1636 nodes and 470 elements; 307 in the panel and 163 in the reinforcement. The mesh was refined in the regions near the junctions where perturbations in the stress field resulting from second order effects were expected based on previous studies [4].

Fig. 7 shows the finite element mesh for the simplified neutral opening. Because this model was prepared after results for the more complex neutral opening were available, mesh generation was simplified since the regions of high stress gradient were known from the results for the previous model.

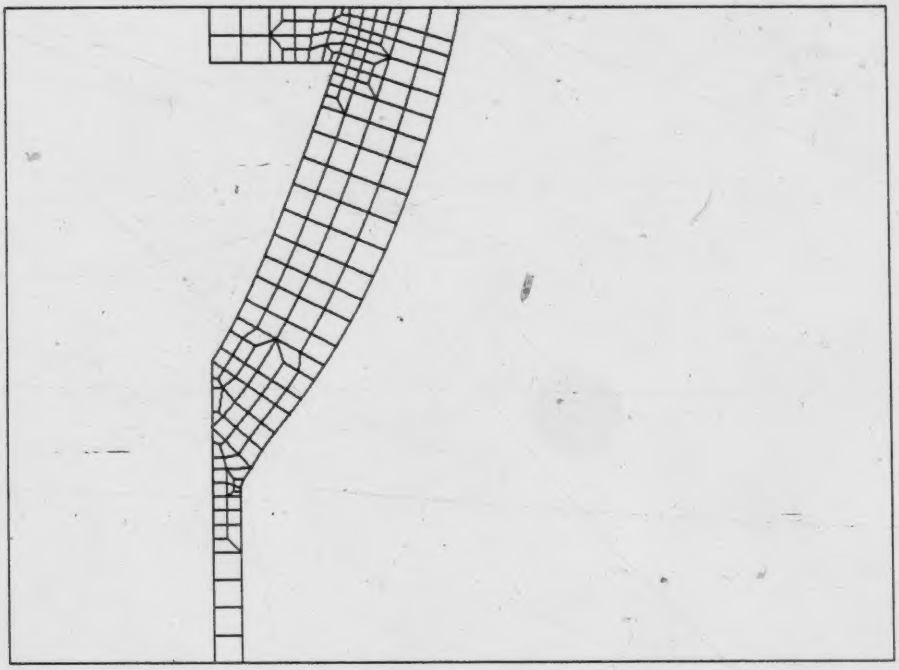


Figure 2-4-a
Bulky Channel Reinforcement
Finite Element Mesh For Liner

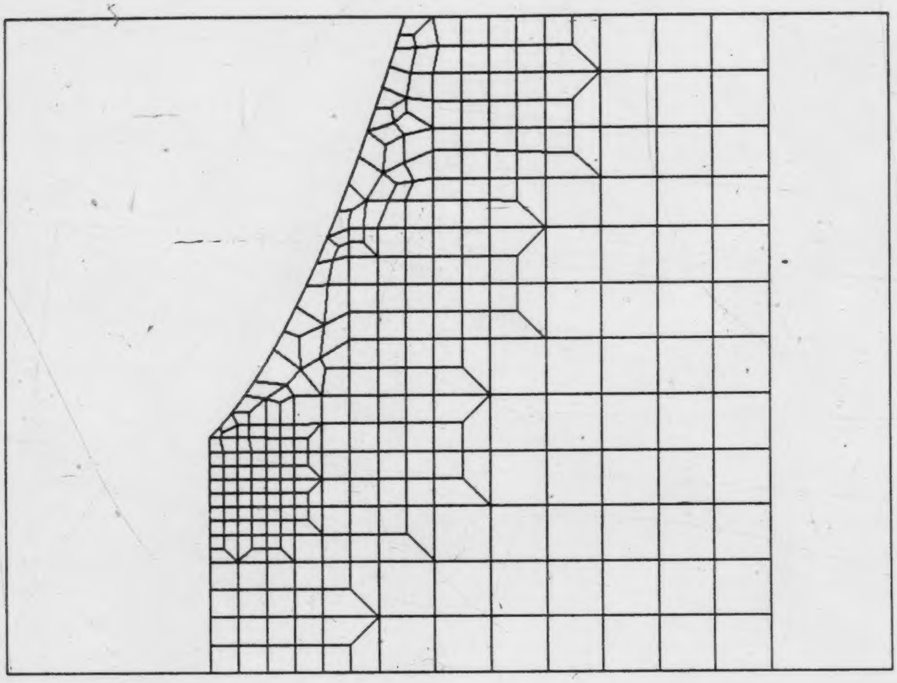


Figure 2-4-b
Bulky Channel Reinforcement
Finite Element Mesh For Panel

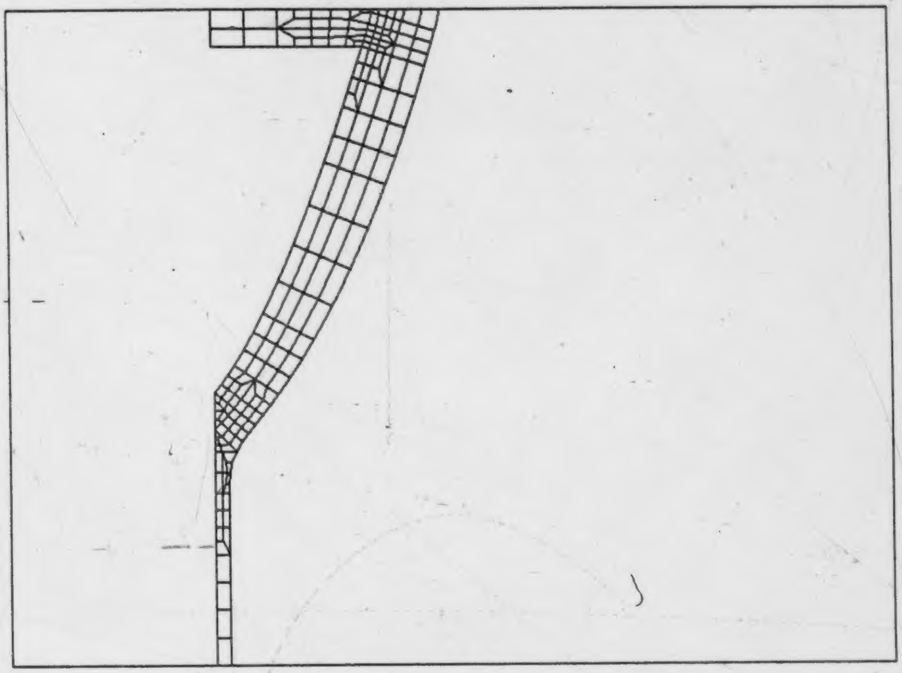


Figure 2-5-a
Compact Channel Reinforcement
Finite Element Mesh For Liner

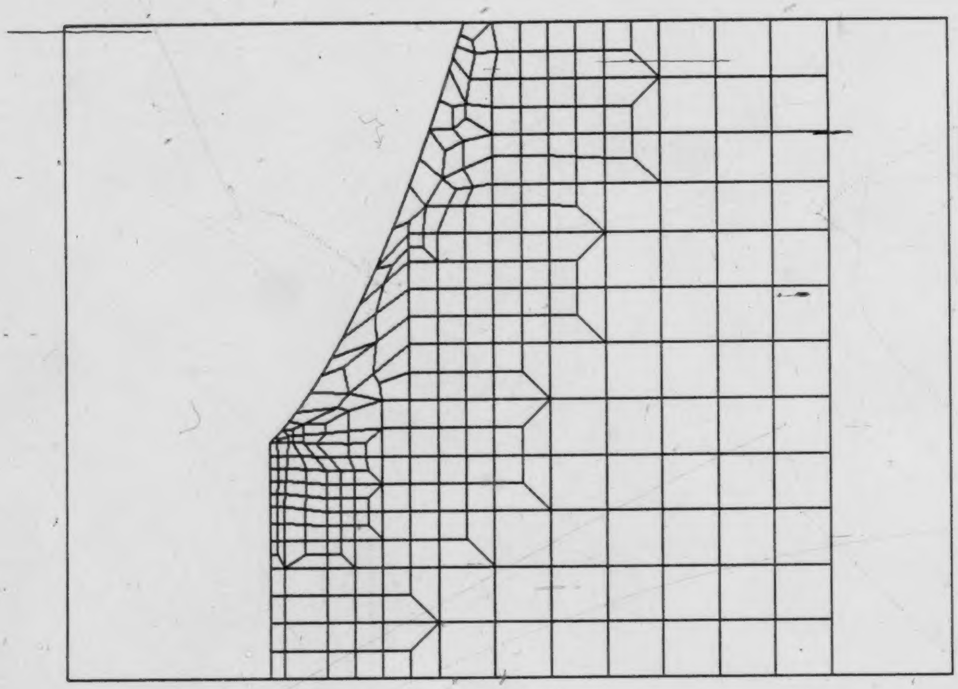


Figure 2-5-b
Compact Channel Reinforcement
Finite Element Mesh For Panel

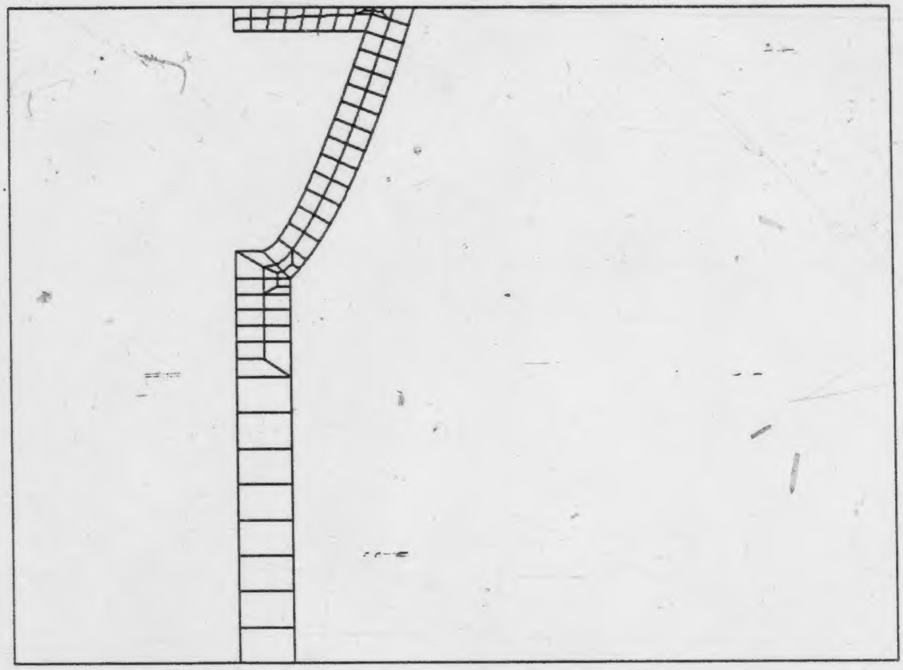


Figure 2-6-a
Simple Reinforcement
Finite Element Mesh For Liner

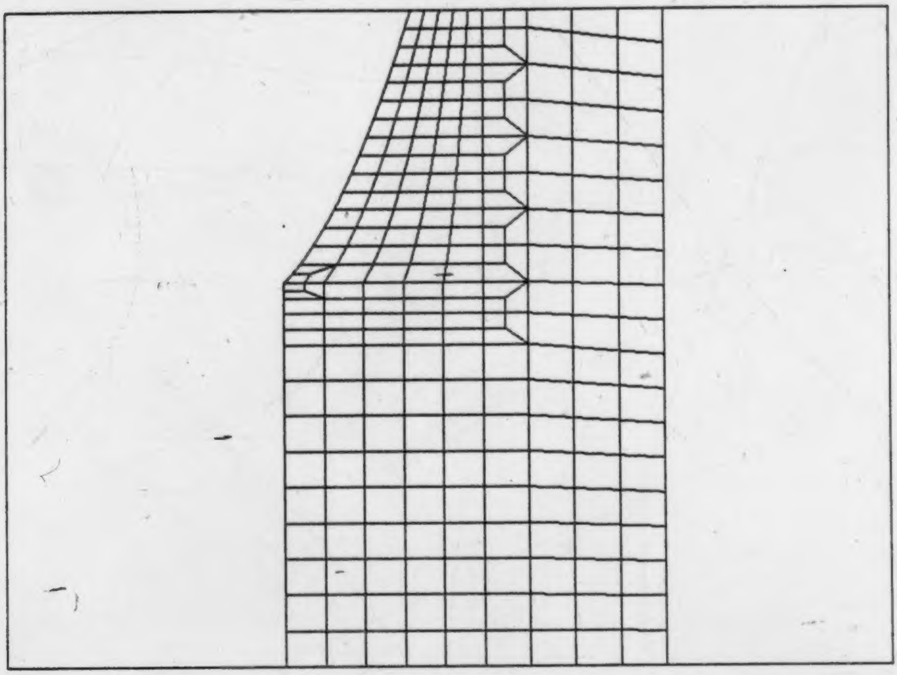


Figure 2-6-b
Simple Reinforcement
Finite Element Mesh For Panel

CHAPTER III

FINEL - FINITE ELEMENT PROGRAM

For Plane Stress, Strain Elasticity Problems

Introduction

The finite element method for solving differential equations is relatively new but, because it offers many advantages over the classical finite difference technique, has become well established as a numerical tool in many fields of study.

In order to produce an accurate model of the continuum, a large number of simultaneous equations is generated requiring solution. For this reason the use of finite element programs has been restricted to mainframe and mini computers. Recently, however, as the capacity and speed of microcomputers has increased, they have become capable of handling some practical finite element programs.

Overview

Finel was developed on an IBM PC personal computer to solve plane stress and plane strain linear elastic problems. The C programming language was chosen for development for three reasons:

1) It produces compact efficient code leaving more room for data.
2) It provides an easy, efficient interface with assembler routines and system utilities.
3) It provides extensive memory management, a prime consideration when using small systems with limited hardware resources.

A special storage technique has been implemented to contain the stiffness matrix so secondary storage is not required and execution time is minimized. As a result Finel, running on a fully configured PC with 640 KBytes of RAM and an 8087 math coprocessor, can solve a problem with approximately 2000 nodes in about four hours. Of course, the exact number of nodes that can be used depends on the specific problem.

Element

The element selected for the program is an eight node quadratic isoparametric element with sixteen degrees of freedom. While such an element produces a more densely populated global stiffness matrix than that generated using a linear displacement element, the mesh required for equivalent accuracy can be more coarse requireing fewer nodes. In addition, the higher order element models curved boundaries much better than the straight sided linear element.

The element uses isotropic, homogeneous, linear elastic material properties. For plane stress problems, the panel thickness is defined at the nodes and interpolated within the element using the interpolation functions. This feature was necessary since the problem to be modelled included a reinforcement which varied in thickness. A detailed development of the element stiffness matrix is included in Appendix A.

Integration of the element matrix is achieved using Gauss quadrature, the order of quadrature being user selectable from two, three, or four point, though two point integration usually yields the best results, as well as improved performance. Zienkiewicz [5] showed that the number of quadrature points required to evaluate the volume of the element exactly is the minimum number needed to ensure convergence upon the correct solution. The integration error introduced using this minimum number actually improves the accuracy of the solution for most problems.

Global Stiffness Matrix

Utilization of memory has been optimized by using a sparse matrix storage method. Because the stiffness matrix is symmetric, only the upper triangular half is stored, reducing storage requirements by nearly one half. The remaining matrix entries are stored as a vector in row major order.

Further reductions are achieved by storing only nonzero matrix entries. To establish where the entry is located in the row, its column number is also stored with the value. The column number of the first (diagonal) entry of each row is the same as the row number, so the column space is used to specify the number of nonzero values in the row.

Because the number of entries varies from row to row, an additional vector must be defined containing pointers to the first entries of the matrix rows. A null row pointer indicates that the degree of freedom is constrained and the row and column associated with it are deleted. Assembly of the elements into this storage scheme is detailed in Appendix B.

Two important features included in the system formation are:

1) Constrained nodal displacements, i.e. one or

both nodal displacements can be described in terms of the displacements of another node.

2) Skewed coordinate systems. These coordinate systems are orthogonal, but are rotated with respect to the global coordinate system.

These features were included to allow a sliding frictionless interface with any slope to be modelled.

System Solver

Because of the rigidly defined matrix structure, a linear equation solver is needed that does not alter the global matrix. Gauss-Siedel iteration was selected as the solution technique with relaxation used to improve the convergence process. For large sparse systems, this method converges on the solution faster than can be achieved using a direct solver. And, because the iterative solution is not sensitive to roundoff error, double precision storage is not required for the stiffness matrix. The system solver and its specific application in Finel are detailed in Appendix C.

Stress Field Evaluation

Once the displacements at the nodes have been solved, the state of stress in the panel can be evaluated. First, nodes using a skewed coordinate system must have their displacements transformed back to x and y components. Then the strains can be calculated anywhere within each element using the interpolation functions.

Because the finite element method minimizes a functional, the state of stress is most accurately evaluated at the Gauss integration points [5]. For this reason, the stress is evaluated at the Gauss points for each element and extrapolated to the nodes using Laplacian extrapolation [7].

Once the stresses have been calculated for all of the elements and the extrapolated values averaged into the nodal values, they are written to disk and can be used directly or used as input for FGRAPH, a program which provides a graphical representation of the stress and displacement field by producing contour plots of the results.

Sample Application of Finel

The following sample problem has been modelled to demonstrate the effectiveness of Finel for solving plane stress problems. The program was used to evaluate the stress field around a hole with a diameter of two units in a panel 30 units square. Utilizing the symmetry of the problem, one quadrant is modelled with normal constraints applied at the lines of symmetry.

The finite element model is shown in Fig. 3-1. The mesh is refined near the hole where the stress gradients are expected to be larger, with ten elements used on the boundary of the opening. The model uses a total of 165 elements with 572 nodes, requiring only one quarter of the program's capacity.

The analytical solution for the stress field around a hole in an infinite panel is well known [6] and was used to provide the loading tractions for the model and as a reference against which the results of the finite element program can be compared.

Fig. 3-2 is a contour plot of the vertical stress produced by interpolating between the nodal values of the analytical solution. Following in Fig. 3-3 are the results of the finite element program. Comparing this with the analytical results, little difference can be seen especially near the hole. An additional contour is seen in the finite element results, but this is in a region where the stress gradient is small and does not represent a significant error.

Fig. 3-4 shows the analytical results near the hole, with the finite element contours in Fig. 3-5. The intersections of the contours with the hole boundary, which are the primary concern, are at nearly the same places in both plots, with a maximum stress concentration of 2.964 being only about 1% in error.

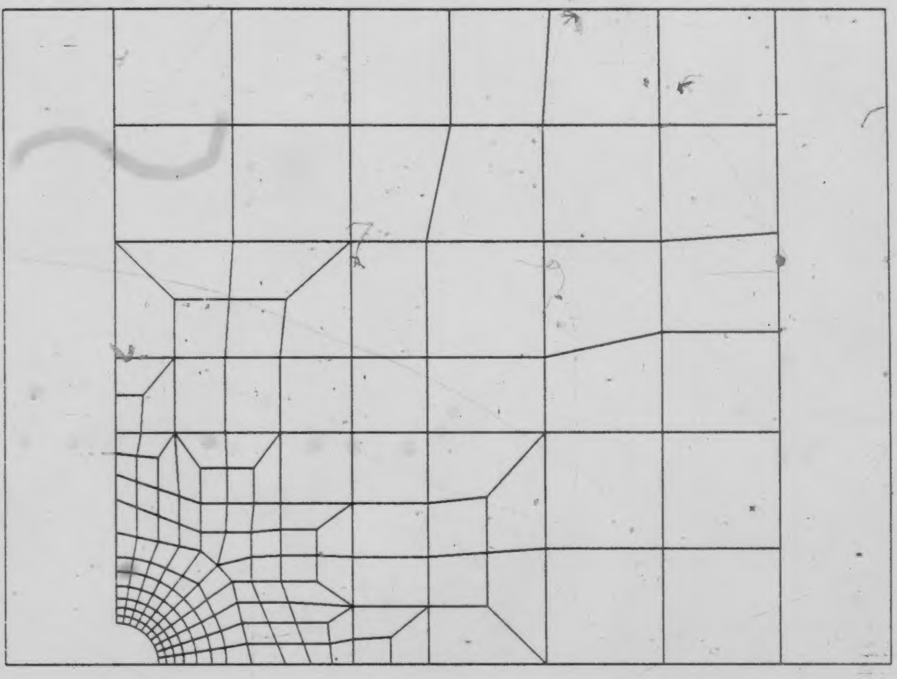


Figure 3-1
Finite Element Mesh
Panel with a Circular Opening

A demonstration finite element model for FINEL, the panel consists of 165 quadratic finite elements defined by 572 nodes, or approximately one quarter of the program's capacity.

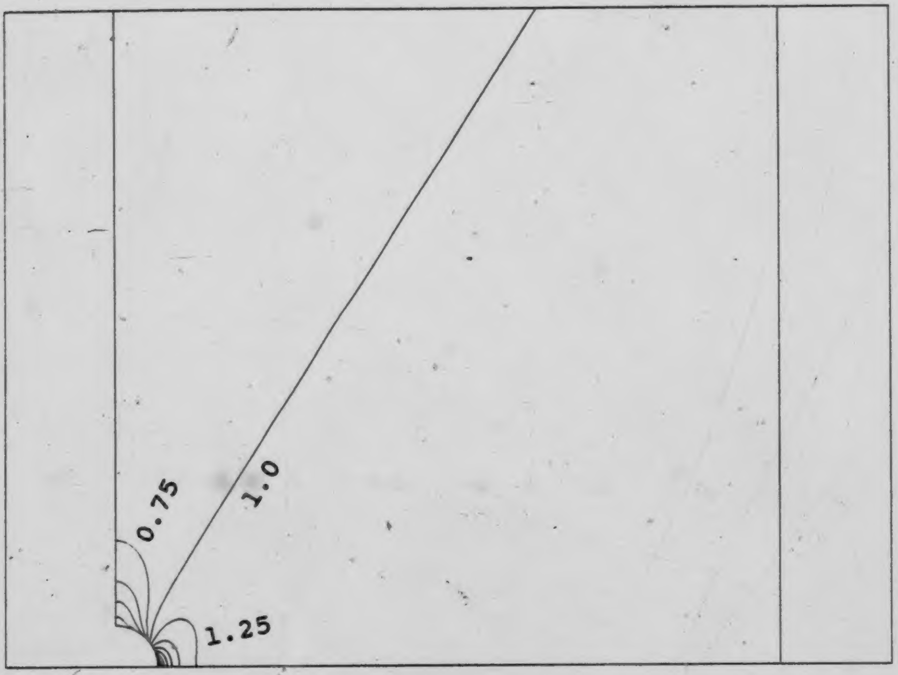


Figure 3-2
Vertical Stress Field - Analytical Solution

The stress concentration of 3.0 is clearly seen on horizontal line of symmetry.

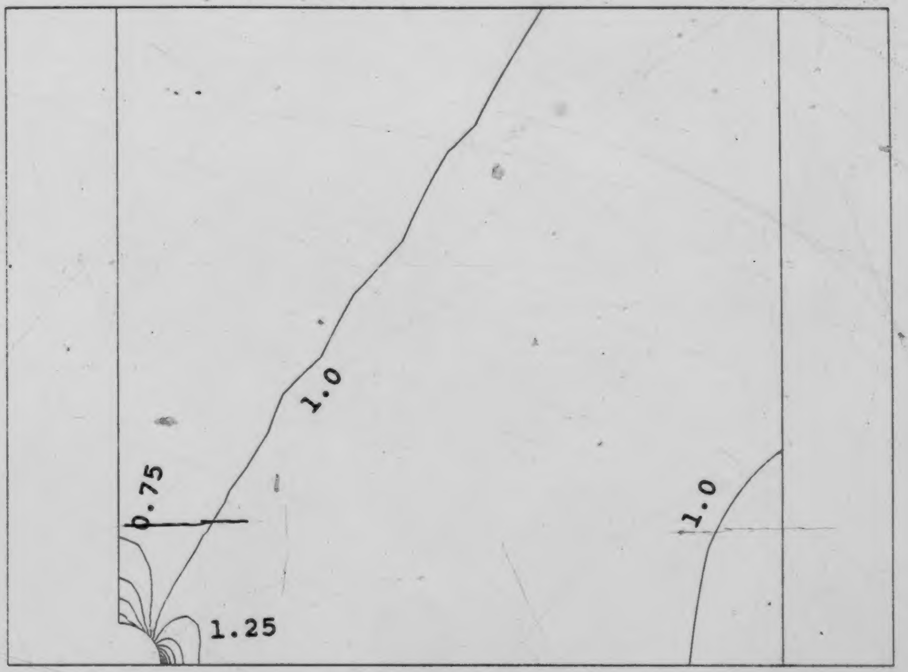


Figure 3-3
Vertical Stress Field - Numerical Solution

Applying loads to the boundary corresponding to the analytical solution of the stress field, The numerical result is nearly identical to the analytical. An additional contour in a region of very small stress gradient is seen but is not significant.

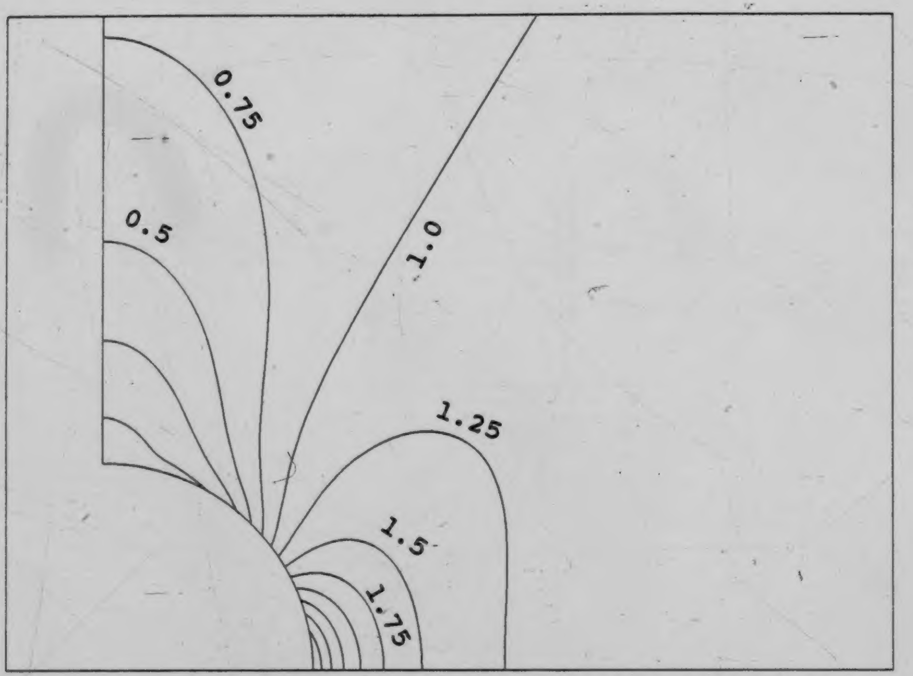


Figure 3-4
Analytical Solution in Region of Opening

σ

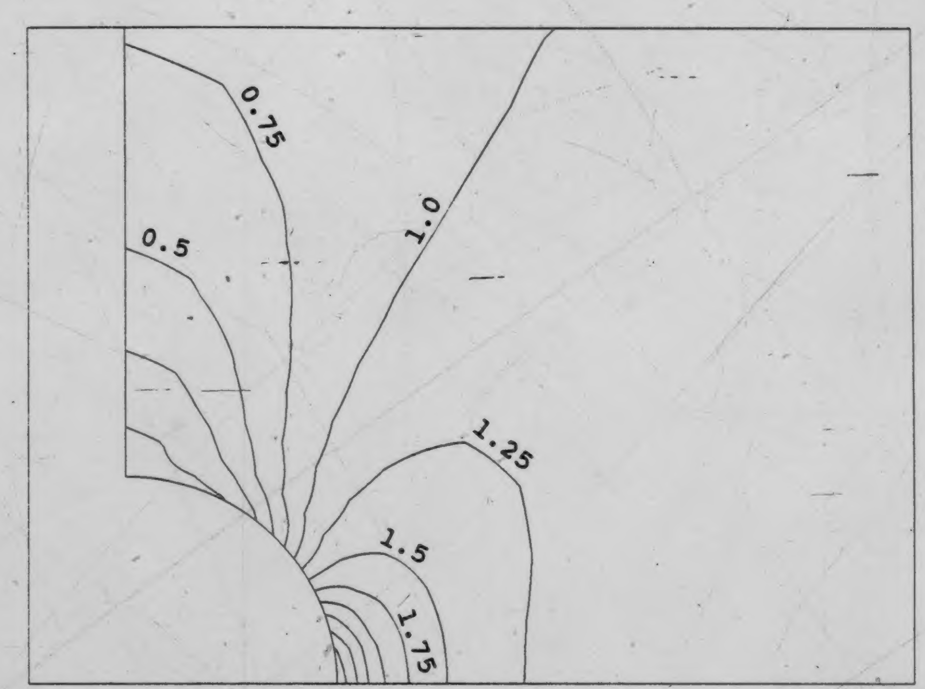


Figure 3-5
Numerical Solution in Region of Opening

Numerical results indicate a stress concentration of 2.964 which is only about 1% in error.

CHAPTER IV

FINITE ELEMENT ANALYSIS RESULTS

General

As described in Chapter 2, all models use the same opening shape. However, selection of different constants for the stress function allows two different reinforcement cross-sections for the channel shaped reinforcement; one relatively square and bulky (B), the other comparatively compact (C), that is, narrow in the plane of the opening shape and deep compared with the panel thickness. Only one stress function was tested for the simple rectangular shaped liner.

For each channel reinforcement, models were analysed with two variations on the junction between the reinforcement and compression member. One model included an integral (I) compression member rigidly attached to the reinforcement and the other featured a compression member insert which was unattached (U) and transferred no shear traction to the reinforcement across the interface which joined them.

In addition to the four models above, all incorporating an integral tension element, one model was analysed using the compact reinforcement and compressive insert along with a tension junction analogous to the compression member insert in that no shear tractions would be transferred from the tension member to the reinforcement at their junction; a junction which can be modelled theoretically, but cannot actually be constructed.

Based on the results of the channel reinforcement analyses, the most effective (practical) junction configuration was selected for the simple neutral opening, that being the compression member insert rather than the integral compression member.

Results of the six analyses are presented as contours maps with uniform increments in values for the selected parameters. The state of stress in the reinforcement is represented by contours of the effective stress (von Mises criterion) with stress concentration increments of 0.1.

Similarly, the stress state in the panel is portrayed by effective stress concentration contours, but because the stress field disturbance is low, stress concentration increments of only 0.025 were used.

The reader is reminded that by utilizing symmetry only one quadrant of the neutral opening required analysis, and the results presented are for that quadrant only.

Behaviour of the panel is also displayed through the use of displacement contours. Using a ratio of x displacement increments to y displacement increments equal to poisson's ratio, a uniform square grid of vertical x displacement contours and horizontal y displacement contours is produced in an undisturbed stress field of uniform tension in the y direction. The disturbance of this uniform grid provides considerable insight into the deformation of the panel in the region of the opening.

It is important to note that this grid of lines is produced by plotting contour lines of constant displacements. It is not a plot of a deformed mesh of regular lines. For example, widely spaced vertical lines indicate a low displacement gradient in the x direction.

Unreinforced Circular Opening

For comparison, contours of effective stress (Fig. 4-1) and displacement (Fig. 4-2) in an infinite panel subject to uniaxial tension in the region of a circular opening are presented. The contour maps represent the solution achieved by the same finite element program used to analyse the neutral openings.

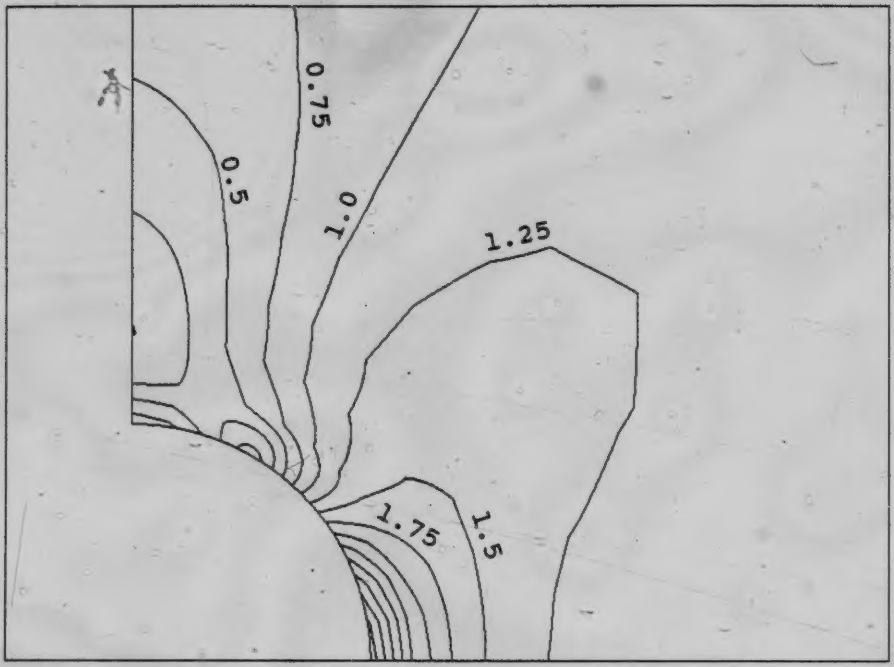


Figure 4-1
Effective Stress Contours about an
Unreinforced Circular Opening.

The Von-Mises effective stress is plotted as a series of contour lines in the neighborhood of the opening.

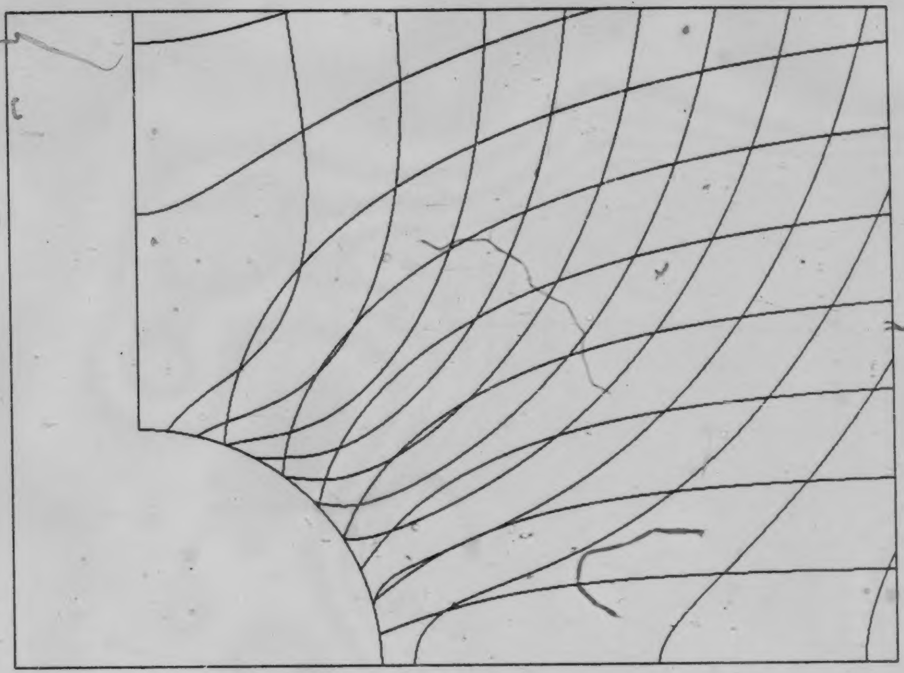


Figure 4-2
Displacement Contour Lines about an
Unreinforced Circular Opening

Contour lines of uniform displacement are plotted for both vertical and horizontal displacement. Contour lines intersecting the right border are vertical displacement contours; those intersecting the top border are horizontal displacement contours.

Bulky Reinforcement with Integral Compression Member (B-I)

Panel

Fig. 4-3 shows a plot of contour lines of constant effective stress in the panel reinforced by the B-I configuration. Considering the small stress concentration increments used, the stress field in the panel is left nearly intact throughout most of the panel.

The high stress gradient seen immediately below the opening is not a stress concentration, but rather a step change in the stress field where the tension member is bonded to the panel. The low stress near the bottom of the opening is a result of the bulkiness of the reinforcement preventing the opening from closing up (Fig. 4-4).

In general, the panel along most of the opening is relatively undisturbed with a stress concentration of about 1.05, but at the junction near the horizontal line of symmetry, a lobe of reduced stress contour lines is seen. This is caused by tension stresses at the bottom of the compression member which prevents the reinforcement from straightening. By restricting bending of the reinforcement, the compression member has caused the disturbance in the stress field of the panel.

The displacement field of the panel produces a fairly square grid of contour lines (Fig. 4-5). Near the bottom of the hole bulging of the vertical lines away from the opening illustrates the effect of the physical bulkiness of the reinforcement at the junction on the stress field. Increased stiffness at the junction prevents the panel from contracting in the horizontal direction resulting in reduced effective stress in that region.

Near the horizontal line of symmetry, the vertical lines (x displacement) are 'pulled' in towards the opening as a result of the compression junction pulling on the reinforcement (superimposed on the compression load on the member, of course).

Reinforcement

The stress field in the reinforcement is represented in Fig. 4-6. The contour lines running approximately parallel to the opening shape indicate a bending moment in the reinforcement as predicted by the neutral opening theory.

Two regions of the plot show severe stress gradients which result from stress concentrations at the junctions. At the bottom of the opening, the V of the opening shape, and hence the reinforcing liner, must close as shown in Fig. 4-4 if the stress field in the panel is not to be disturbed. In a similar fashion, the opening shape opens up at the top of the opening, and the compression junction is expected to open in the same manner.

If the reinforcement sections could pivot at the junctions but still transfer the equilibrating tension, shear and bending loads to the junction, the neutral opening model would be satisfied at the junctions. However, because the behaviour at these intersections is, in effect, bending of a curved beam of very short radius of curvature, a high stress concentration arises.

A close up view of the compression junction is seen in Fig. 4-7. As expected, the stress concentration is located at the notch between the compression member and reinforcement. Separate analysis (results of which are not presented here) showed that rounding this notch did little to reduce the stress concentration at this location but instead moved the stress riser down the reinforcement to the point where the filet began. It is concluded, therefore, that the major cause of the stress concentration is not the existence of the notch but instead the physical bulkiness of the reinforcing members.

Finally, Fig. 4-8 shows an enlarged view of the tension junction. Here, the stress concentration of 2.27 is not located in the region of a sharp notch and is caused by the closing action of the junction.

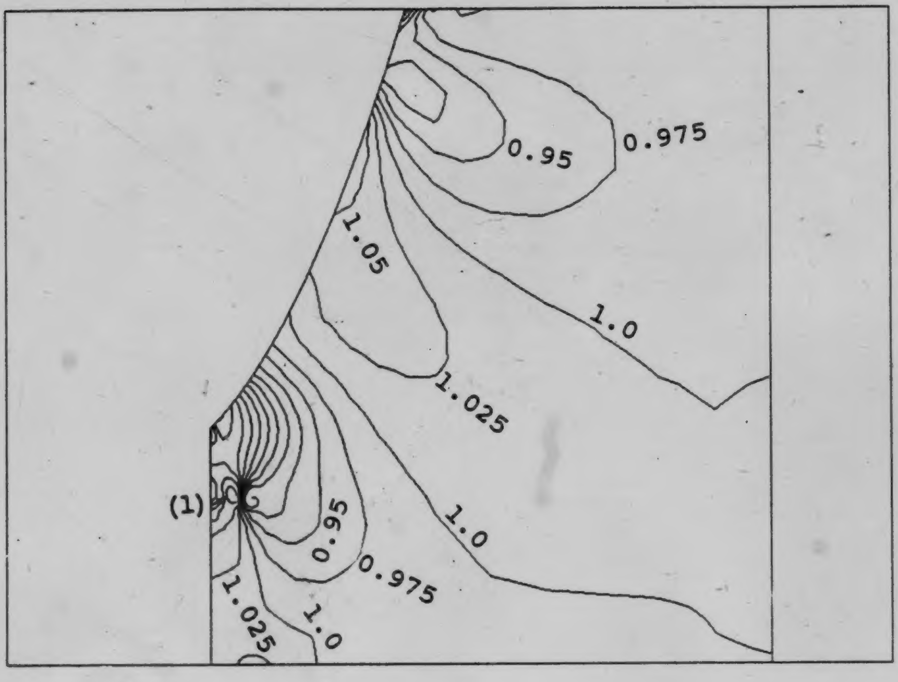


Figure 4-3
B-I Reinforcement
Effective Stress Contours for Panel

- Effective stress increments for contours = 0.025
- (1) High stress gradient where tension member is attached, not a stress concentration.
- Lobe of contours at bottom of opening is a region of low effective stress produced by the junction with the tension member.

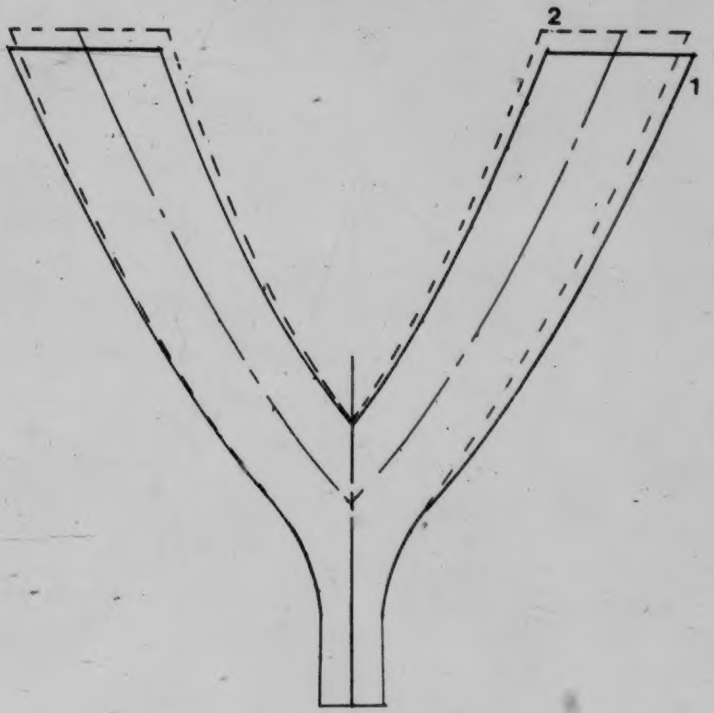


Figure 4-4
Tension Junction Behaviour

Originally, the opening has a configuration as shown in (1). When loaded with tension, the panel will become longer and narrower and the opening will assume the configuration in (2). The tension junction resists the closing of the junction and the stress in the region of the junction is reduced from nominal.

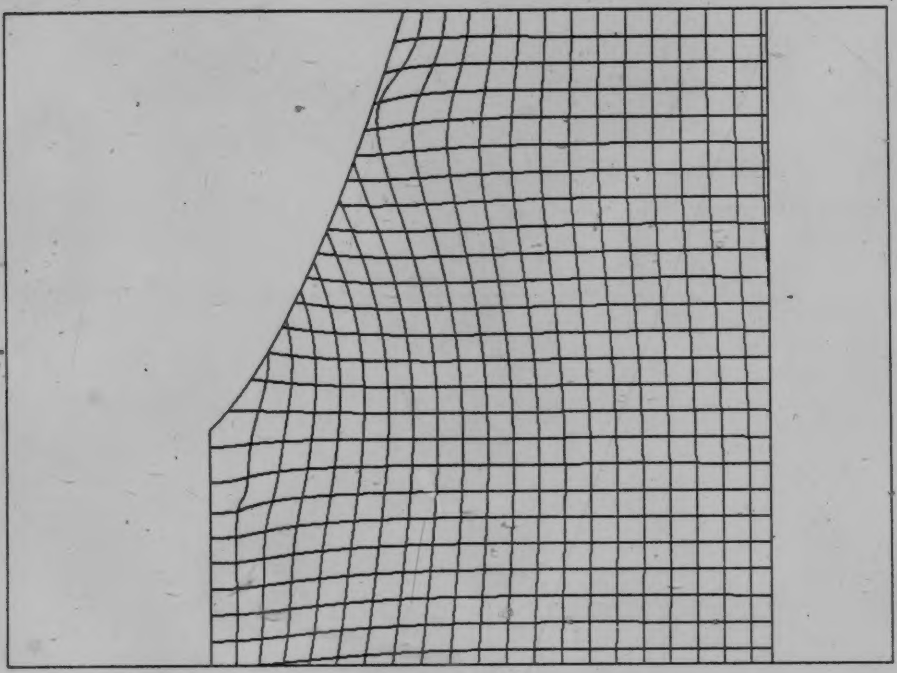


Figure 4-5
B-I Reinforcement
Displacement Field in Panel

Vertical lines near bottom of opening bulge away from hole because of bulkiness of the junction.
Vertical lines near horizontal line of symmetry are pulled towards liner because attachment to compression member prevents bending of liner.

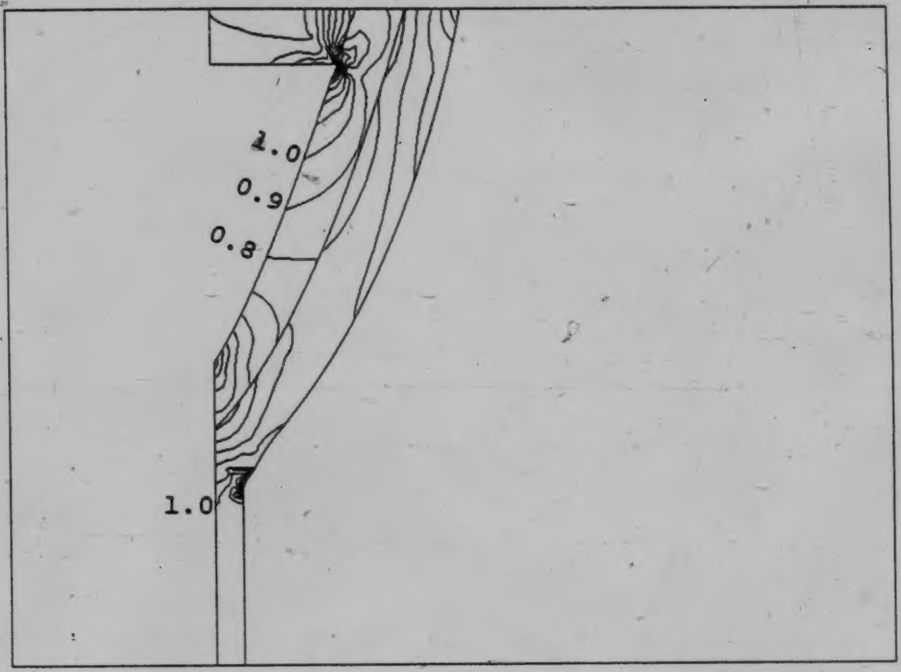


Figure 4-6
B-I Reinforcement
Effective Stress Contours for Reinforcing Assembly

A uniform stress field is seen through most of the liner and some bending is indicated. Stress concentrations are indicated are indicated at each of the junctions.

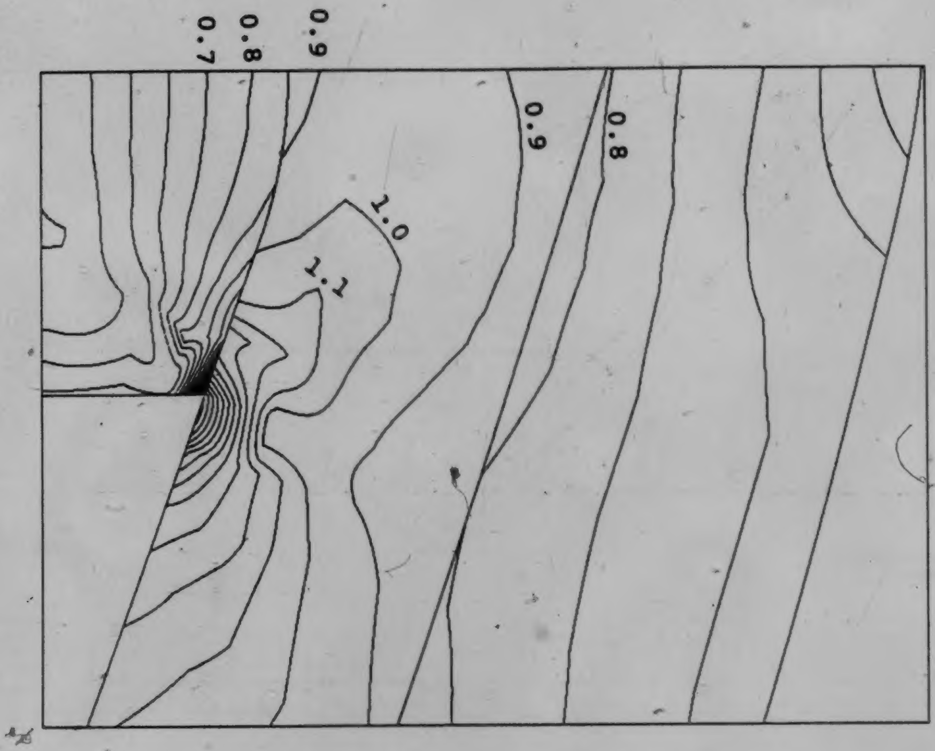


Figure 4-7
B-I Reinforcement
Effective Stress Contours in Compression Junction

A stress concentration of 2.27 is indicated at the notch in the junction. Separate analysis indicate that the notch is not responsible for the stress concentration.

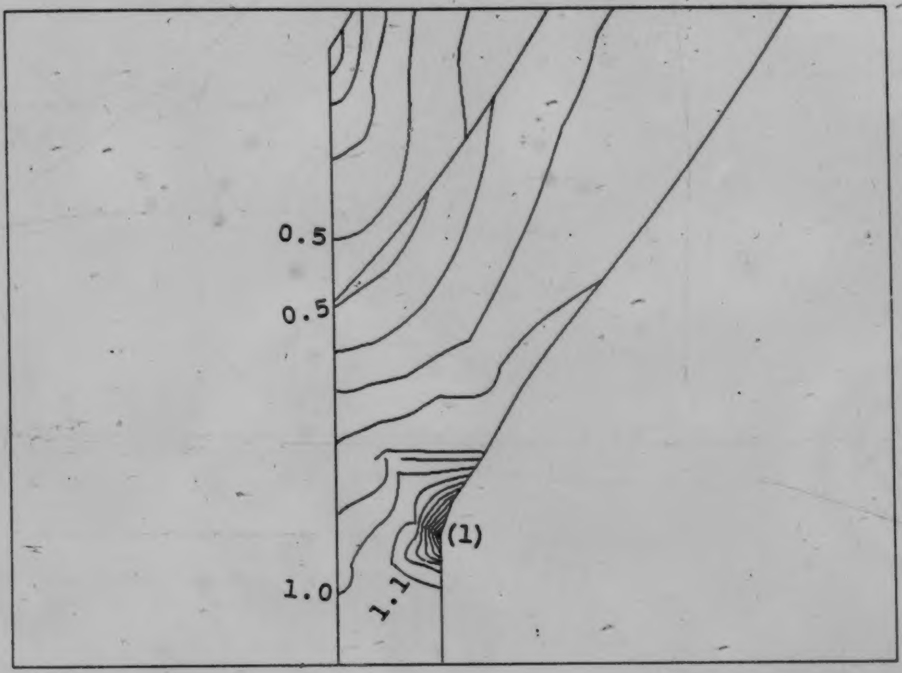


Figure 4-8
B-I Reinforcement
Effective Stress Contours in Tension Junction

A stress concentration of 1.93 is indicated at point (1). No notch is present here, so the stress concentration is caused by the bulky junction.

Bulky Reinforcement with Compression Member Insert (B-U)

Panel

Figure 4-9 shows the stress field in the panel near the opening for the model with the unattached compression member. Though similar to Fig. 4-3 near the bottom of the opening, this plot shows the stress field to be slightly more uniform along the midsection of the opening, and much less disturbed near the horizontal line of symmetry, with the lobe of contour lines eliminated almost completely. Some disturbance is still apparent at the line of symmetry, arising from the stress concentration in the reinforcement which has been relocated (to be discussed in the next subsection).

The improved stress state in the panel is further demonstrated by the contour map of displacements in Fig. 4-10. Bulging of vertical contour lines is still apparent near the bottom of the opening, but overall a much more uniform grid is seen here than in the previous displacement contour map, Fig. 4-5, due to the reduced stiffness in the junction.

Reinforcement

The sliding interface at the compression junction has not significantly altered the overall stress distribution in the reinforcement, Fig. 4-11. Compared with Fig. 4-6, however, the stress concentration has shifted up to the horizontal line of symmetry. Fig. 4-12, an enlarged view of the compression junction, shows that the boundary of the reinforcement at the interface has been specified to be a circular arc starting approximately one half of the way down the interface and curving normal to the plane of symmetry as it intersects it, eliminating any notch at the symmetry line. The fact that a stress concentration persists indicates the problem is a result of the physical bulkiness of the reinforcing members not considered by the neutral opening theory.

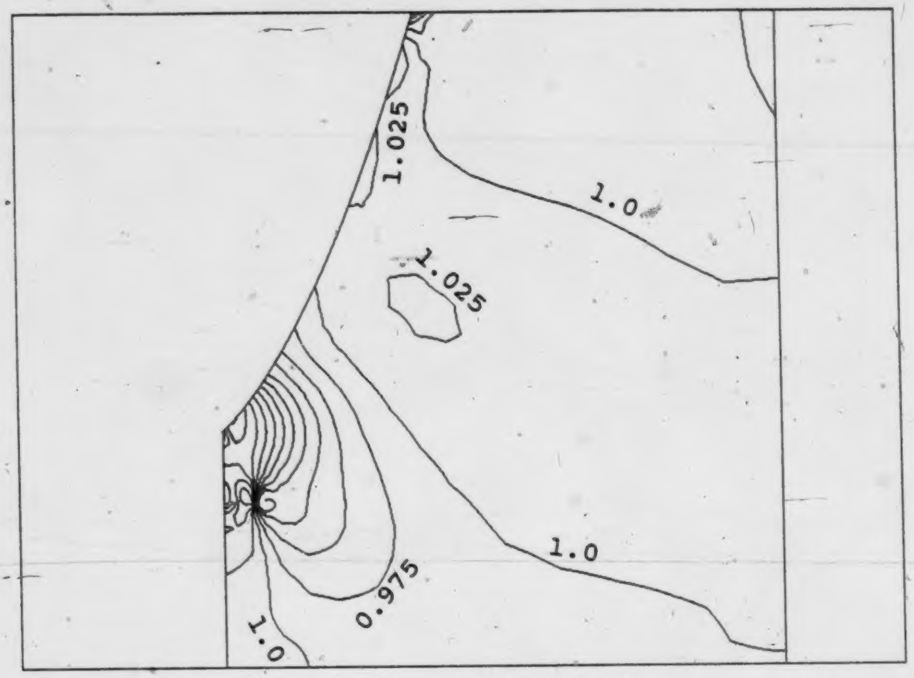


Figure 4-9
B-U Reinforcement
Effective Stress Field in Panel

Stress field is more uniform than B-I reinforced panel, especially near the compression junction at the horizontal line of symmetry where the disturbance caused by restiction of the reinforcement in bending is almost eliminated.

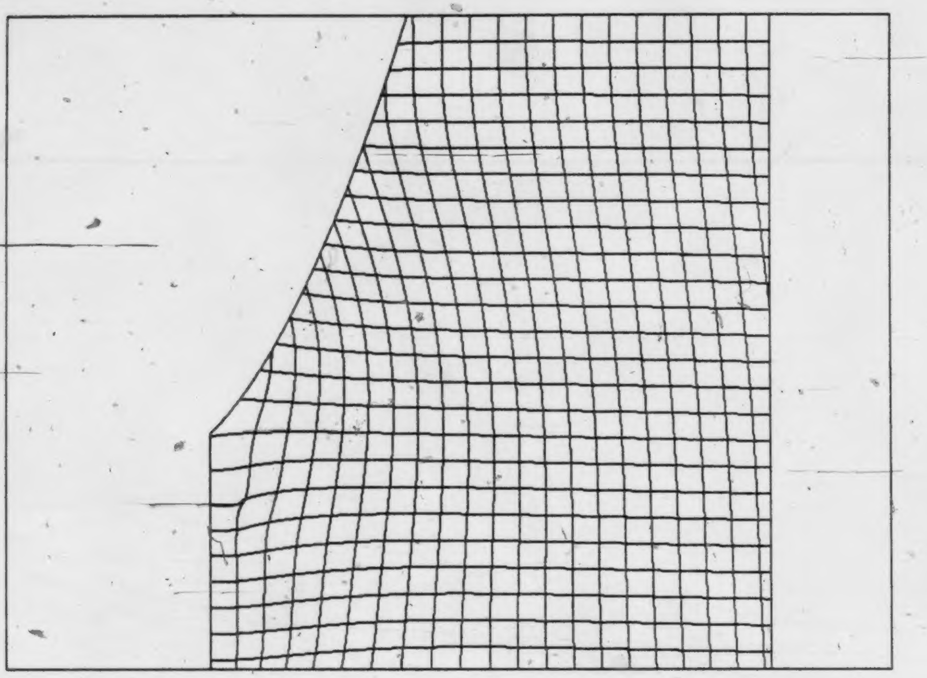


Figure 4-10
B-U Reinforcement
Displacement Field in Panel

Square grid of displacement contour lines indicates a nearly undisturbed stress field in most of the panel. Most of the disturbance is seen near the bottom of the opening where the tension junction restricts deformation and lower stresses result.

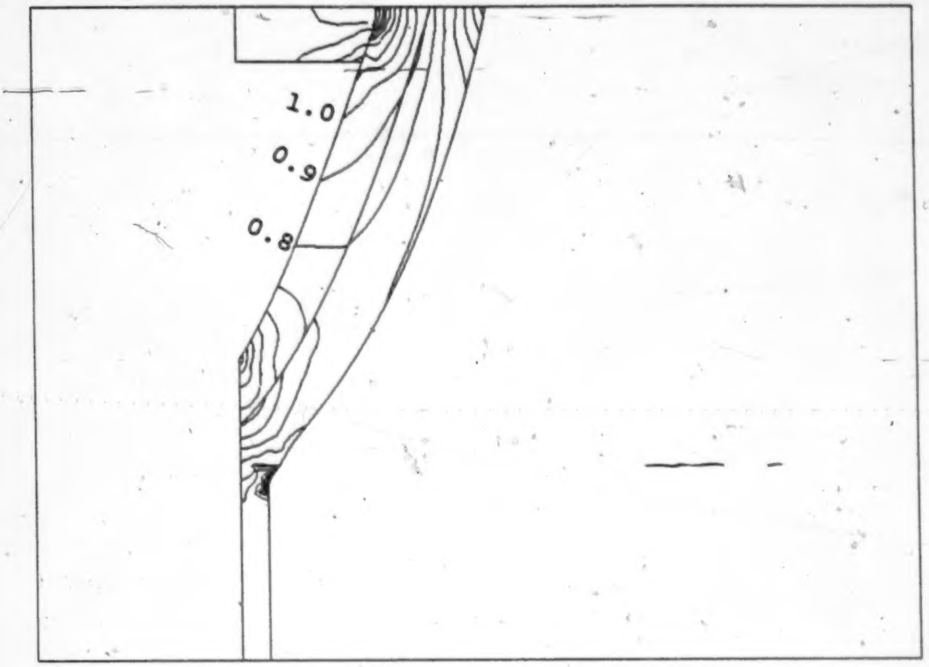


Figure 4-11
B-U Reinforcement
Stress Field in Reinforcing Assembly

Stress concentration in compression junction is located on the horizontal line of symmetry.

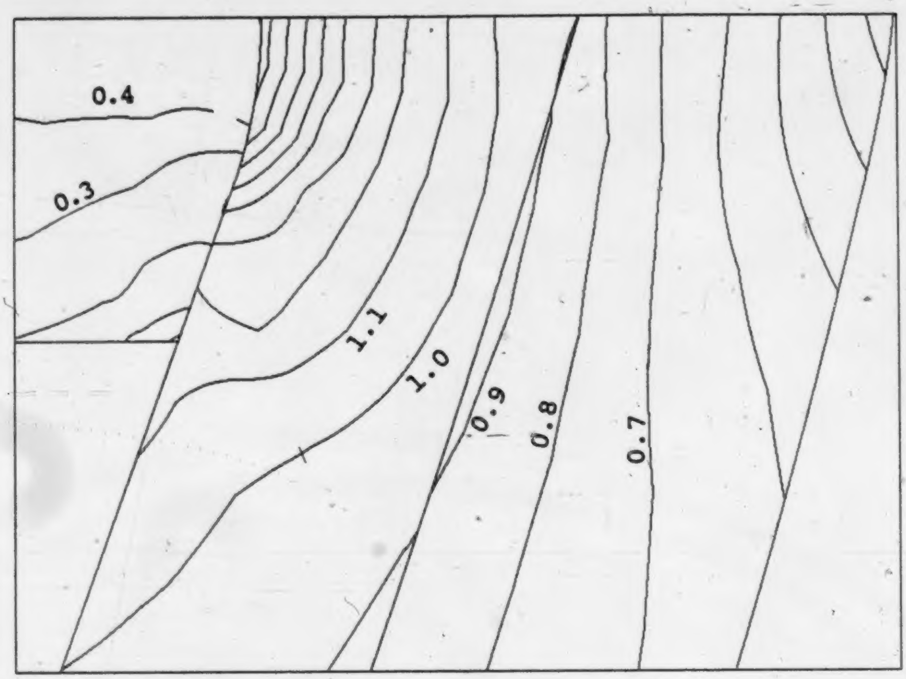


Figure 4-12
B-U Reinforcement
Stress Field in Compression Junction

Stress concentration of 1.86 is indicated on the horizontal line of symmetry in the reinforcement (1). A circular boundary has been described for the reinforcement to eliminate any notch that may exist.

Compact Reinforcement with Integral Compression Member (C-I)

Panel

Figure 4-13 shows the stress field in the panel reinforced by the compact reinforcement. The contour map is very similar to that produced when the bulky reinforcement is used (Fig. 4-3), the only significant difference being the slight vertical displacement of the lobe of contours near the horizontal plane of symmetry corresponding to the narrower (though deeper) compression member.

In similar fashion, the displacement contours in Fig. 4-14 display the same characteristics as Fig. 4-5, with the disturbance near the horizontal line of symmetry displaced upwards because of the narrower compression member.

Reinforcement

The stress field in the reinforcement, Fig. 4-15, shows fewer contour lines than the B-I model. These contours are roughly parallel to the opening shape. A lower gradient normal to the opening, along with the narrower reinforcement, indicates a significantly reduced bending moment compared with the bulky reinforcement in Fig. 4-6.

It was expected that a more compact reinforcement would reduce the stress concentrations at the tension and compression junctions. In general significant reductions were achieved in the tension junction and nominal reductions were realized for the integral junction. The compression junction, Fig. 4-16, is not significantly different from Fig. 4-7 and, although reduced by 0.54, the maximum stress concentration is still 1.73.

At the tension junction, Fig. 4-17, the stress field produces a plot for which the contours have a form slightly different from Fig. 4-8, which is a result of the differences in geometries between the two junctions. The maximum stress concentration factor in this junction has been lowered by 0.60 to a value of 1.33.

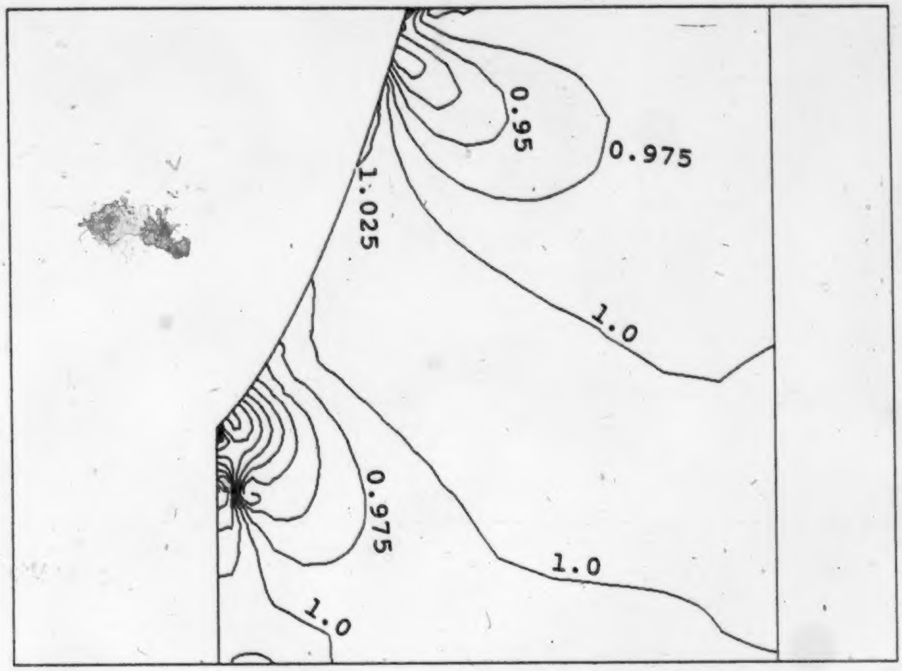


Figure 4-13
C-I Reinforcement
Stress Field in Panel

Similar to B-I reinforced panel (4-3) except that lobe of contours near vertical line of symmetry is slightly higher corresponding to the narrower, deeper compression member.

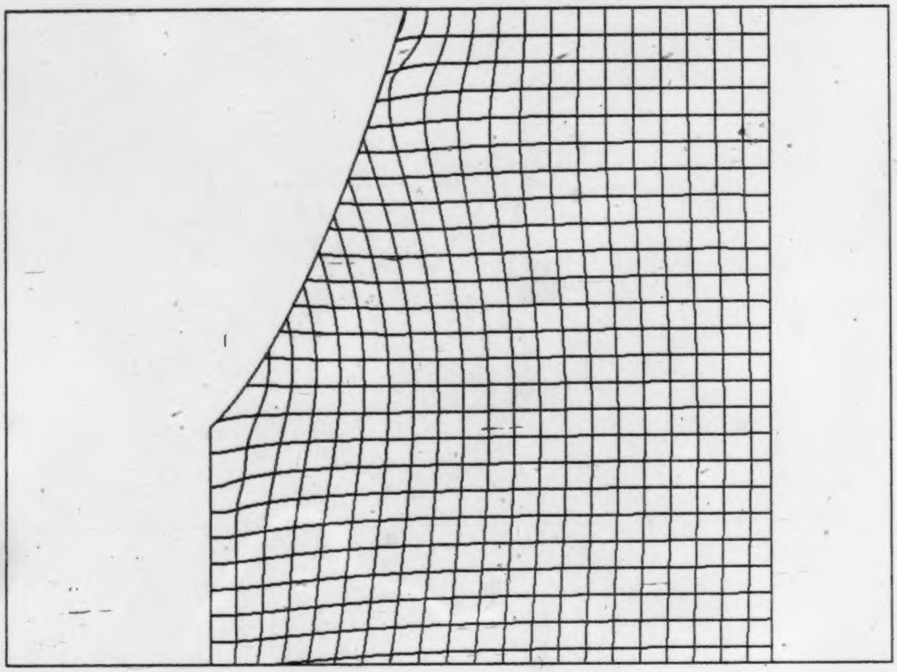


Figure 4-14
C-I Reinforcement
Displacement Field in Panel

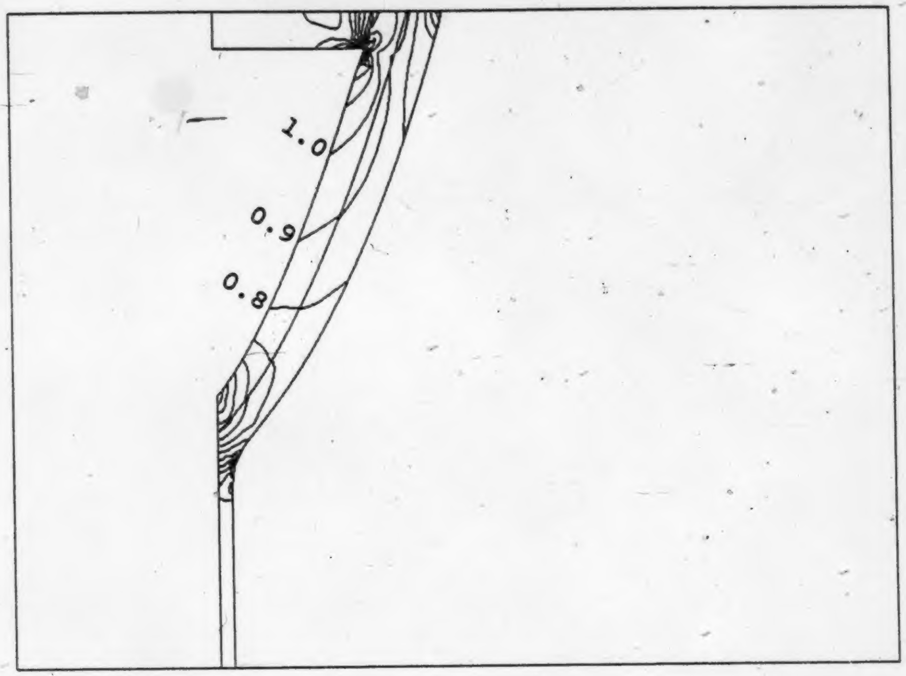


Figure 4-15
C-I Reinforcement
Stress Field in Reinforcement

Fewer contour lines in this reinforcement are apparent indicating the reduced bending moment. Bending concentrations at the junctions are still significant.

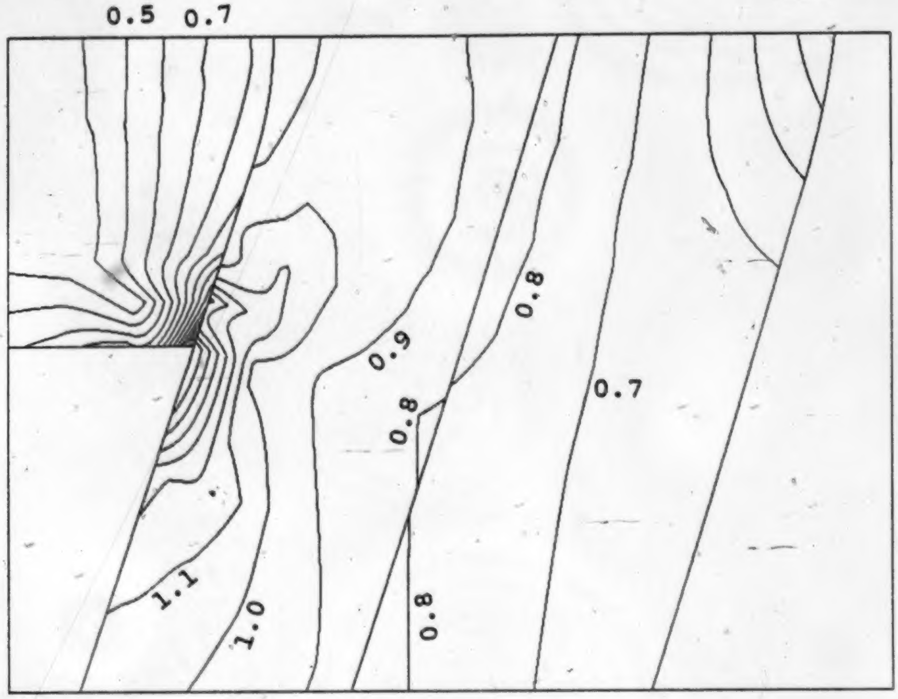


Figure 4-16
C-I Reinforcement
Stress Field in Compression Junction

Stress concentration of 1.73 is large, but a significant reduction from 2.27 experienced by the B-I reinforcement.

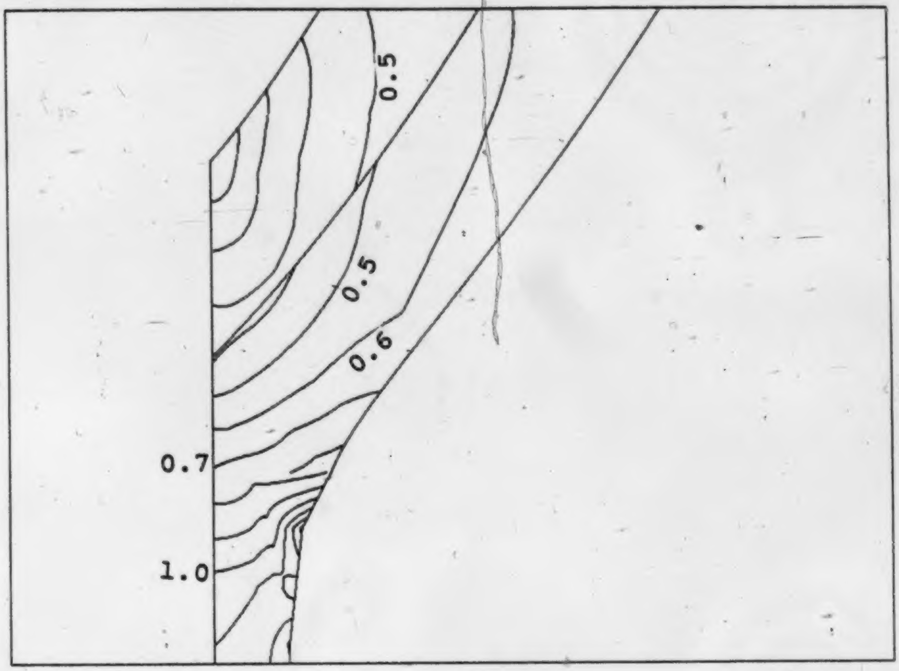


Figure 4-17
C-I Reinforcement
Stress Field in Tension Junction

The maximum stress concentration is 1.33, significantly lower than the value of 1.93 in the bulky model.

Compact Reinforcement with Compression Insert (C-U)

Panel

The stress field for this model, Fig. 4-18, shows the same improvement in the state of stress in the panel over Fig. 4-14 that was seen under the same compression junction conditions in the bulky reinforcement. The contour map of Fig. 4-18 is, as a result of this improvement, barely distinguishable from Fig. 4-9, the B-U reinforced panel. The displacement contours, Fig. 4-19, also are nearly identical to those produced by the bulky model, Fig. 4-10, which indicates that the compact reinforcement provides no real advantages over the bulky reinforcement as far as the panel stress field is concerned.

Reinforcement

Figure 4-20, containing contours for the overall reinforcement stress field, demonstrates once more that the new compression junction does not significantly affect the stress state of most of the reinforcement.

The compression junction itself, Fig. 4-21, exhibits the same change in the contour map as the bulky reinforcement did in Fig. 4-12. Unlike the integral compression junction, where the compact reinforcement provided at least a small reduction in the maximum stress concentration, the stress riser for this model has a maximum value of 1.94, which is 0.21 greater than the compact reinforcement with integral compression member but 0.33 less than the same junction in the B-U model. Clearly, though more compact, this model still possesses a physical bulkiness which gives rise to the stress concentration.

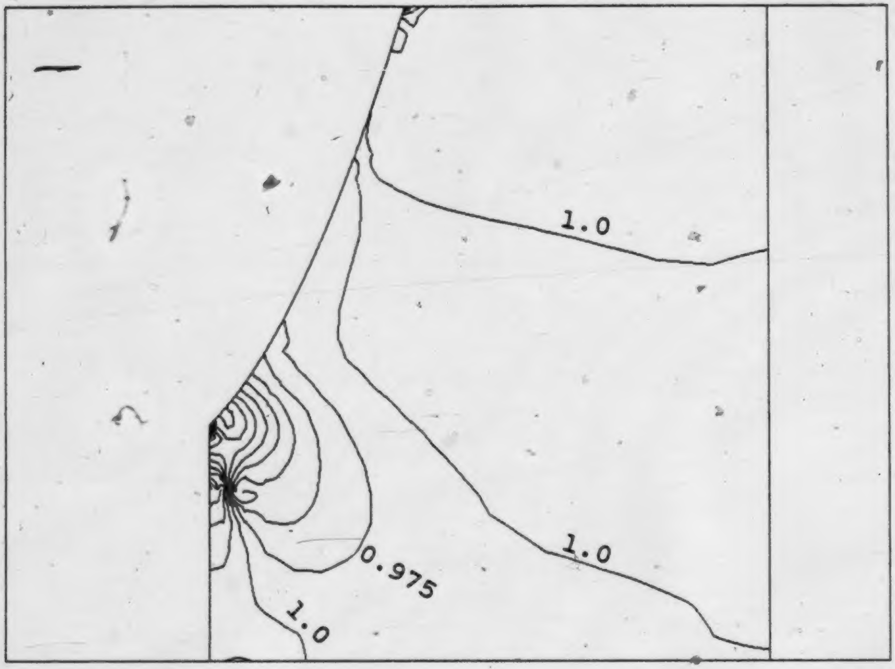


Figure 4-18
C-U Reinforcement
Stress Field in Panel

The same improvement in the stress distrubution is seen in the panel with a compact reinforcement when the compression member is left unattached at the junction.

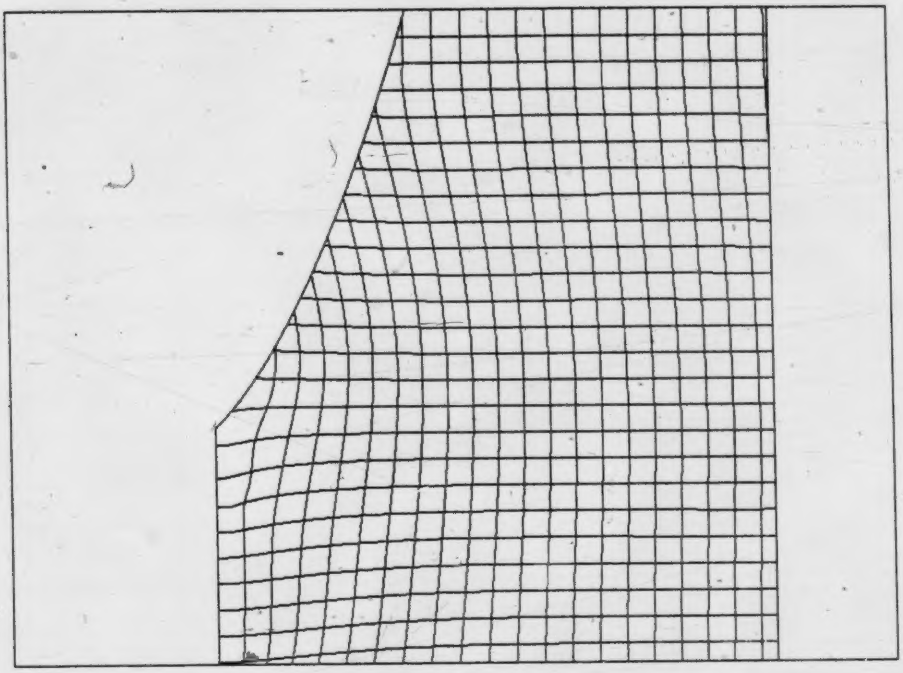


Figure 4-19
C-U Reinforcement
Displacement Field in Panel

This distribution is almost identical to the B-U reinforced panel (4-10). This indicates that both reinforcements are equally capable of leaving the stress field in the panel undisturbed.

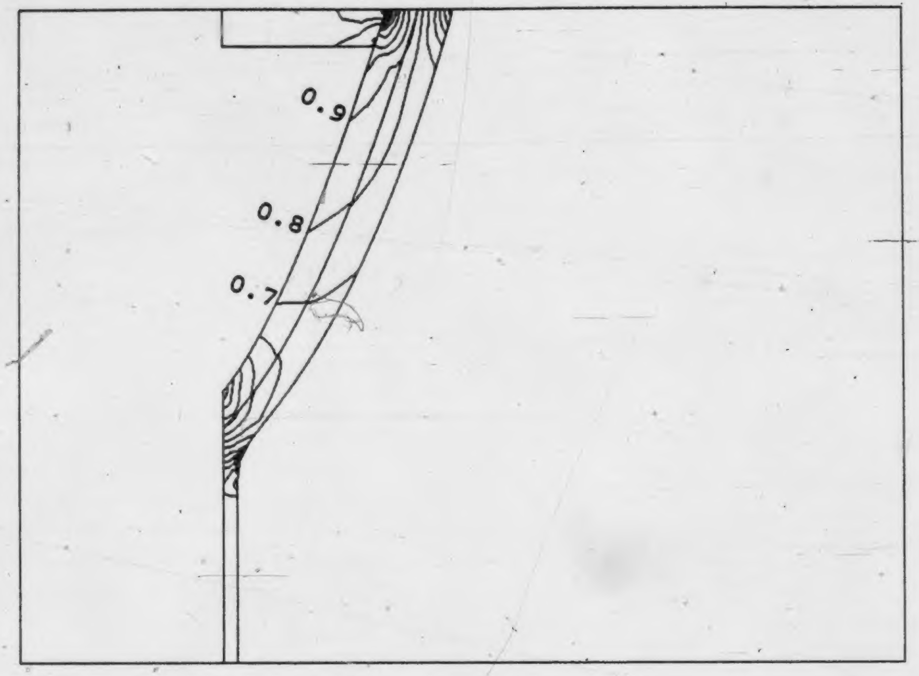


Figure 4-20
C-U Reinforcement
Stress Field in Reinforcement

As in the bulky reinforcement, the only significant difference in stress fields between the integral and unattached models is in the location of the stress concentrations.

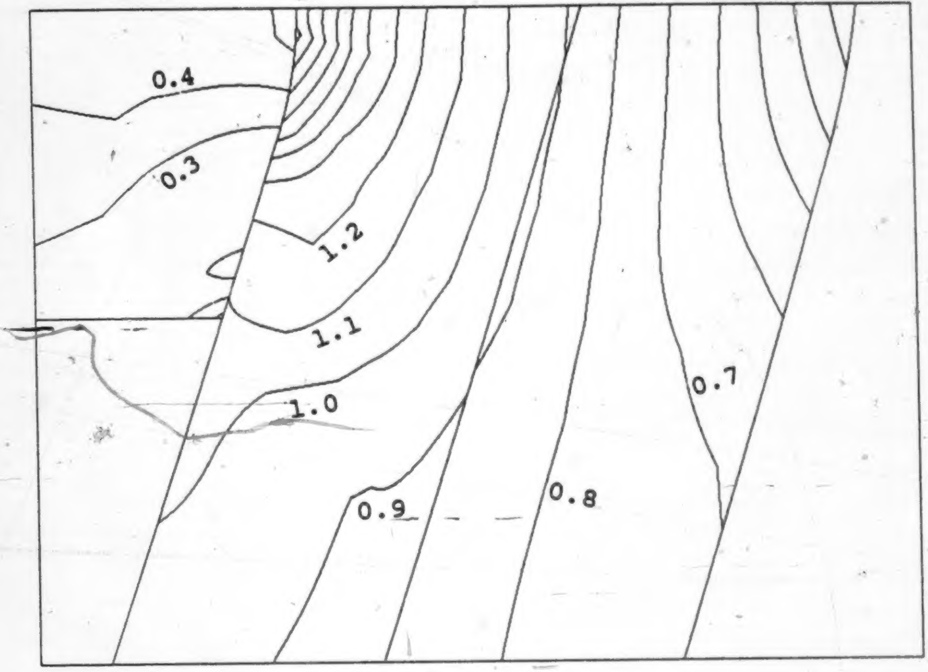


Figure 4-21
C-U Reinforcement
Stress Field in Compression Junction

The maximum stress concentration of 1.94 is higher than the integral junction but still lower than the bulky than the unattached bulky junction. It is also very effective for leaving the panel undisturbed.

Compact Reinforcement with Sliding Interface at Tension Junction (C-U-U)

Panel

The stress field in the panel, Fig. 4-22 shows the same pattern of contour plots as Fig. 4-18 for the C-U model, with a slight improvement near the bottom of the opening. The lobe of contours has been reduced in size, and the lowest effective stress has been increased to 0.73 from 0.70 indicating a slightly more uniform stress field in this area.

These small differences in stresses result from very small changes in displacement. The displacement contours in Fig. 4-23 are virtually identical to those found in Fig. 4-19.

Reinforcement

The overall stress distribution in the reinforcement is essentially unchanged from the solid tension junction. The stress field in the junction itself, as shown in Fig. 4-24, is considerably different as a result of the sliding interface between the reinforcement and tension member. Although this configuration improves the stress field in the panel, the maximum stress concentration in the tension junction is increased by 0.25 to a value of 1.58.

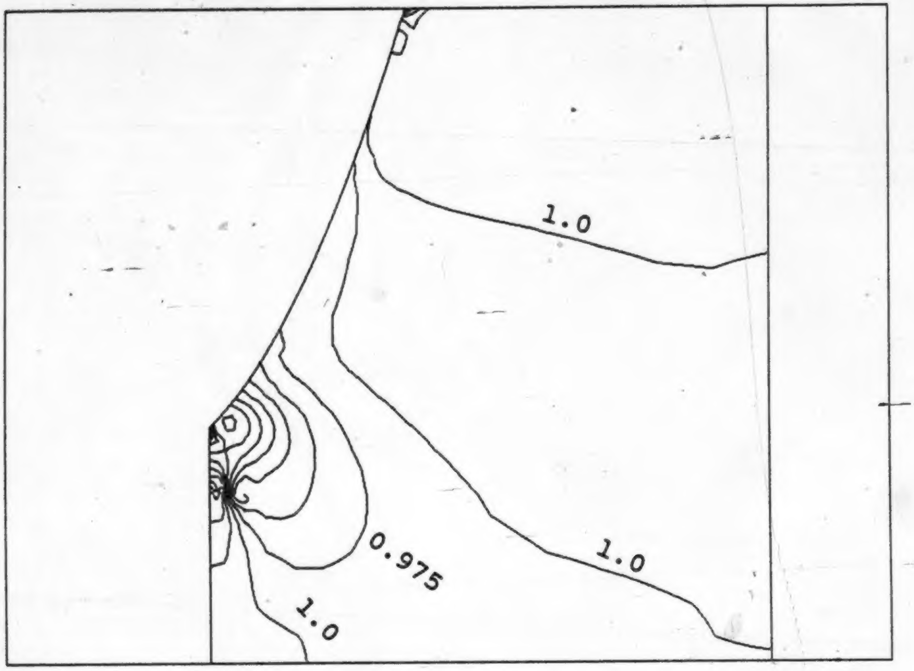


Figure 4-22
C-U-U Reinforcement
Stress Field in Panel

The hypothetical tension contact junction results in a slight improvement in the stress field in the panel.

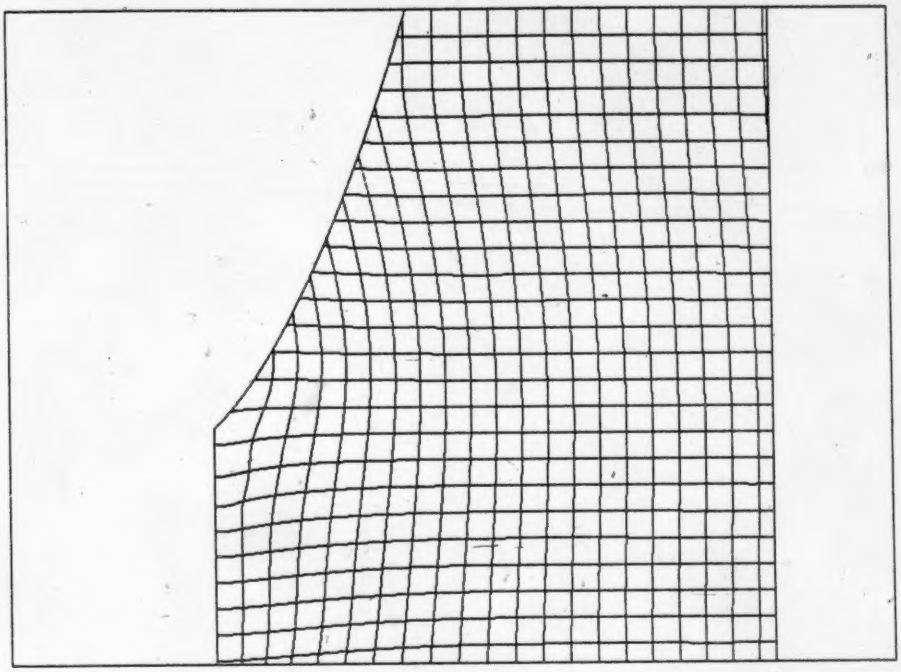


Figure 4-23
C-U-U Reinforcement
Displacement Field in Panel

The displacement field in the panel with a contact tension junction is virtually identical to that with an integral tension junction.

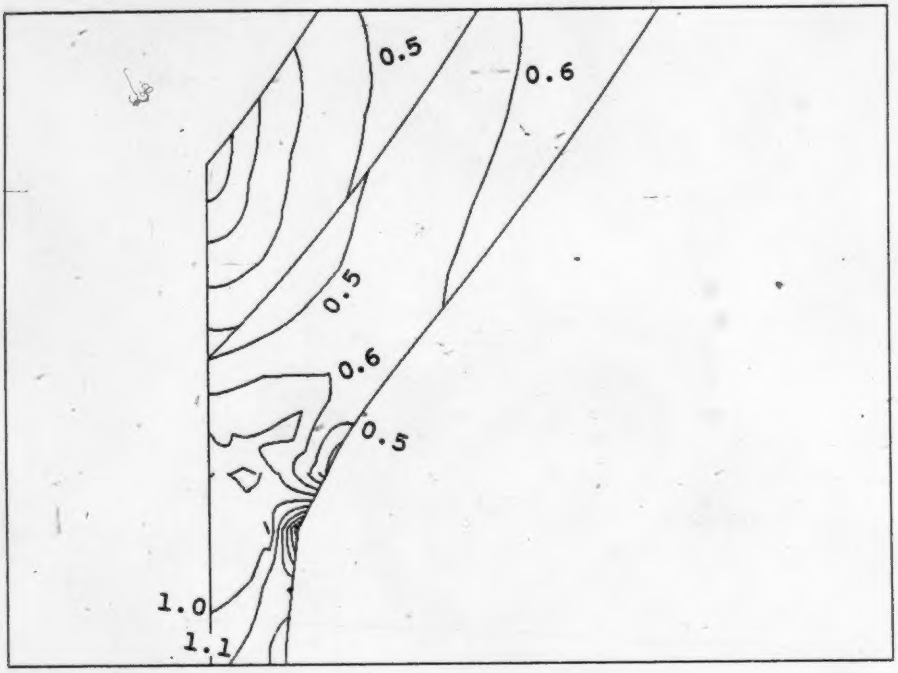


Figure 4-24
C-U-U Reinforcement
Stress Field in Tension Junction

Although the stress field in the panel is slightly improved, stress concentration in the junction jumps by 0.25 to a value of 1.58.

Simple Rectangular Reinforcing Liner

Panel

Figure 4-25 portrays the stress field in the panel reinforced by the simple rectangular reinforcement (S-U). Near the bottom of the opening, where the stress was reduced by the junction stiffness in other models, the stress increased in this simple assembly. The maximum stress at this location is 1.48. This arises because the panel is now attached to the outside fibre of the reinforcement where the stress concentration in the junction exists rather than to the neutral axis, as was the case in the previous models.

Compared to previous models, the stress in the panel is reduced at the top of the model since the panel is attached to the region of the reinforcement where the stress is reduced by bending of the liner. This ~~region~~ was well removed from the panel in the other models.

The displacement contours (Fig. 4-26) show the panel to be considerably more disturbed than for previous models, but better than for an unreinforced circular hole (Fig. 4-2).

Reinforcement

The stress contours in the simple reinforcement, shown in Fig. 4-27, indicate that the liner behaves in the same manner as the channel reinforcements. The interface at the compression junction (Fig. 4-28) was not rounded and contributes to the high stress concentration of 1.78. This could probably be reduced somewhat, but would have very little effect on the stress distribution in the panel.

A stress concentration of 2.79 is apparent at the tension junction (Fig. 4-29). The extra stiffness of the panel bonded to the junction at the location of the maximum stress concentration was expected to reduce the stress concentration more effectively than it did.

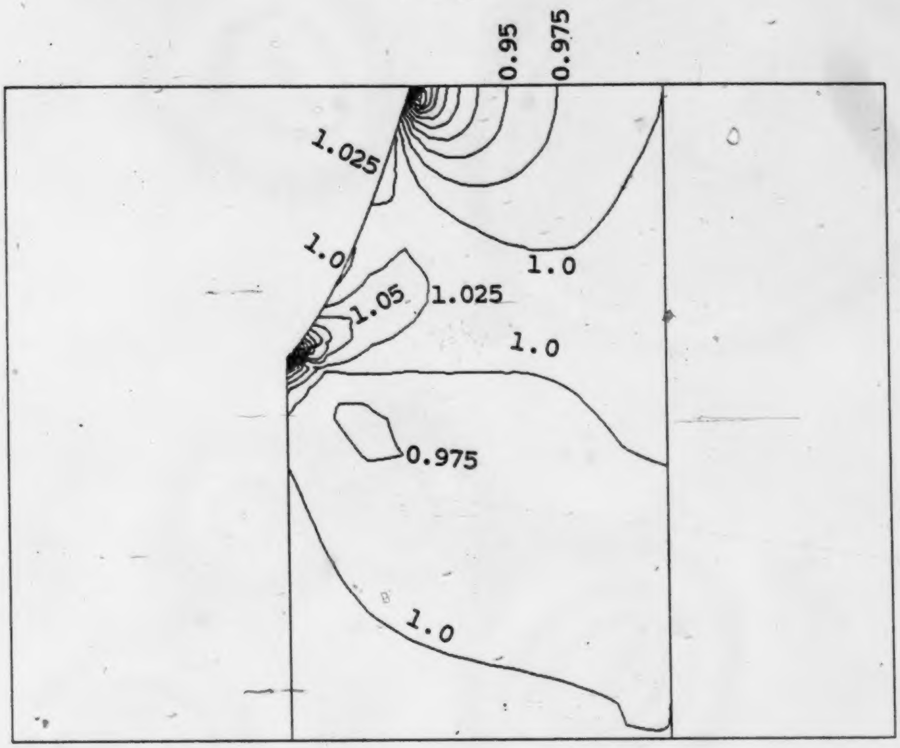


Figure 4-25
S-U Reinforcement
Stress Field in Panel

A stress concentration in the panel at the tension junction arises because liner is not fastened at neutral axis. This also results in a reduced stress at the compression junction.

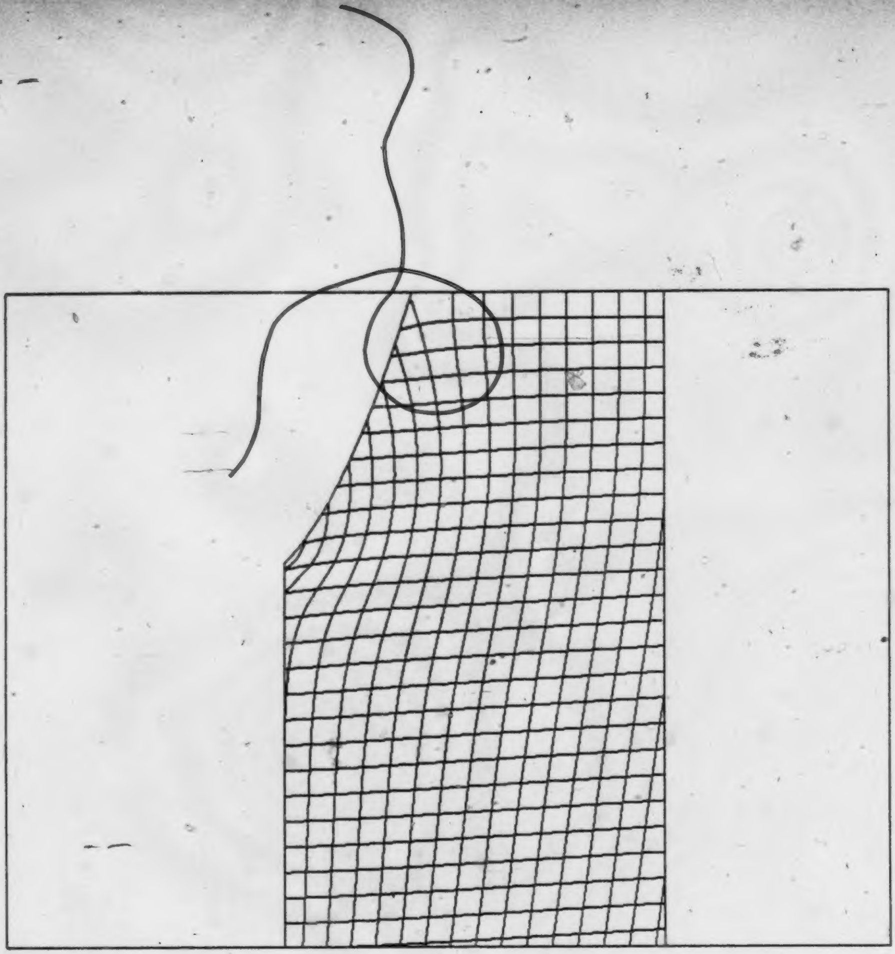


Figure 4-26
S-U Reinforcement
Displacement Field in Panel

The displacement field shows the panel to be disturbed by the reinforced opening. However the disturbance is small compared to the unreinforced opening (4-2)

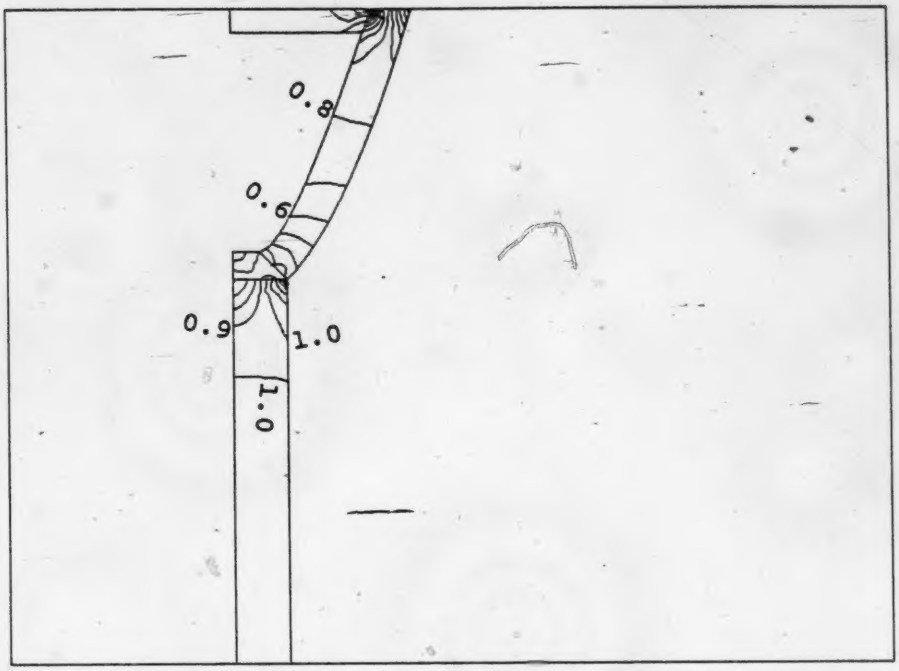


Figure 4-27
S-U Reinforcement
Stress Field in Reinforcement

Stress distributions in the simple recangular reinforcement are similar to those in the more sophisticated channel shaped reinforcement.

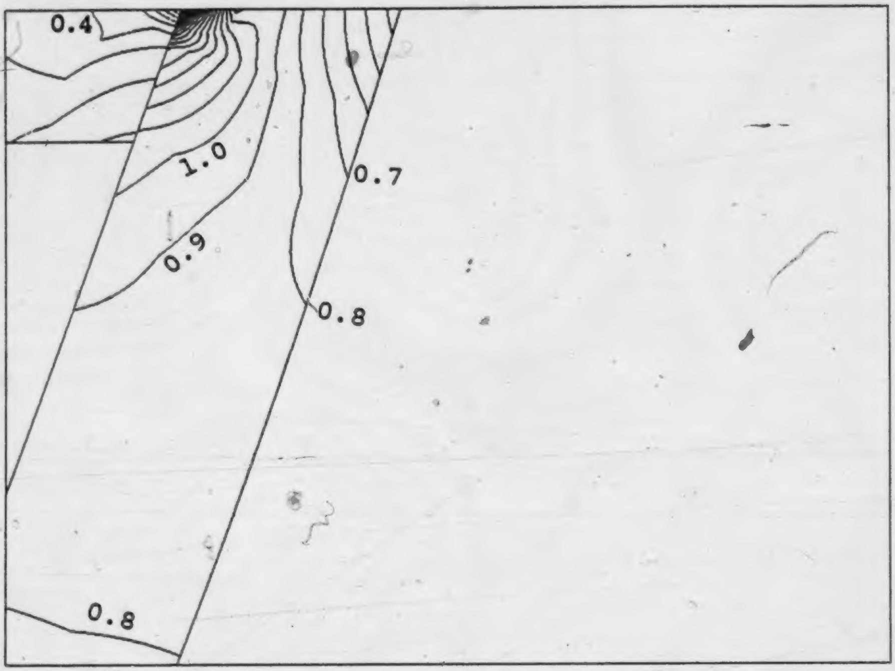


Figure 4-28
S-U Reinforcement
Stress Field in Compression Junction

An unrounded interface contributes to the high stress concentration of 1.78. Removing the notch reduces this value only slightly and has no effect on the stress field in the panel.

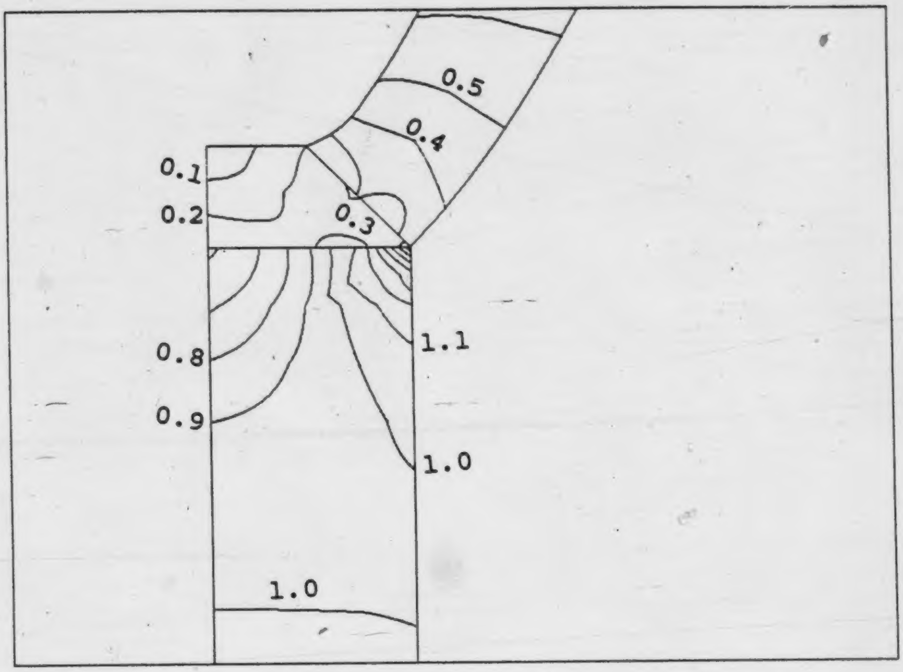


Figure 4-29
S-U Reinforcement
Stress Field in Tension Junction

The additional stiffness of the panel bonded to the extreme fiber of the reinforcement did little to reduce the stress concentration. In fact the stress concentration was increased to 2.79.

Summary

Results of the analyses of various neutral openings are summarized in table 4-1. Stress concentration factors are tabulated for several key locations in each panel and reinforcement

The table shows that the compact reinforcement with sliding interfaces at the junctions is most effective for producing an undisturbed stress field in the panel with maximum and minimum stress concentration factors closest to unity.

The compact reinforcement is shown to produce the lowest peak concentration factors. This indicates that reinforcements approaching the Mansfield neutral opening are more desirable.

The simple geometry reinforcement is shown to be much less effective than the channel shaped reinforcement for producing an undisturbed stress field.

Table 4-1

LOCATION	B-I	B-U	C-I	C-U	C-UU	S-U
Panel						
- max.	1.10	1.09	1.10	1.09	1.09	2.79
- min.	0.68	0.68	0.70	0.70	0.73	0.01
- opening max.	1.07	1.05	1.04	1.02	1.02	1.04
Compress.						
- max.	2.27	1.86	1.73	1.94	1.94	1.78
Tension						
- max.	1.93	1.93	1.33	1.33	1.58	2.79
- min.	0.11	0.12	0.10	0.12	0.73	0.09

B-I - Bulky Liner with Integral Compression Member

B-U - Bulky Liner with Unattached Compression Member

C-I - Compact Liner with Integral Compression Member

C-U - Compact Liner with Unattached Compression Member

C-UU - Compact Liner with Unattached Compression Member & Sliding Tension Junction

S-U - Simple Rectangular Liner with Unattached Compression Member

CONCLUSIONS

Neutral opening theory has been extended to include bending and shear considerations for the reinforcing liner. Additionally, the theory has been generalized so that it may be applied to any opening shape for any given stress function. The result is a set of expressions for the cross sectional characteristics of the liner (the area, moment of inertia and bulk shear factor) in terms of the opening shape and stress function only.

Generally, one or more of the cross sectional characteristics are undefined or physically unacceptable at one or more locations for a proposed opening shape. A closed shape may still be formed from a piecewise continuous opening shape with reinforcing members attached to equilibrate the unbalanced loads at the junctions. New expressions for these equilibrating reinforcing members at the junctions were also derived to account for the shear and tensile loads.

Earlier studies of compact reinforcement neutral openings indicated perturbations of the stress field in the panel adjacent to the junctions. However, the photoelastic techniques could not provide any information about the stress field in the junctions of the reinforcement.

In this study the effectiveness of the theory was tested using the finite element method to model an opening with several reinforcing configurations in a panel subject to uniaxial tension. The reinforcing material was the same as that in the panel. This numerical technique provided details of the stress field in the panel and in the reinforcing assembly. Determination of the stress field within the junctions was of considerable interest. The results show that, although the perturbation of the stress field in the panel does comply with neutral opening theory, stress concentrations in the junction can be high, though generally not as great as those in an unreinforced opening.

REFERENCES

1. Mansfield, E. H., "Neutral Holes in Plane Sheet - Reinforced Holes Which Are Elastically Equivalent to the Uncut Sheet," Quarterly Journal of Mechanics and Applied Mathematics, Vol 6, Part 3, 1953, pp 370-378.

2. Richards, R. and Bjorkman, G. S., "Neutral Holes: Theory and Design," ASCE, Vol. 108, No. EM5, October, 1982.

3. Rogers, G. L., and Causey, M. L., Mechanics of Engineering Structures, John Wiley & Sons, Inc., 1962, pp. 246-252.

4. Budney, D. R., and Bellow, D. G., "On the Analysis of Neutral Holes", Experimental Mechanics, Vol. 22, No. 9, Sept. 1982, pp. 348-353.

5. Zienkiewicz, O. C., The Finite Element Method, Third Edition, McGraw-Hill, 1983.

6. Timoshenko, S. and Goodier, J. N., Theory of Elasticity, Second Edition, McGraw - Hill, 1951.

7. Huebner, K. H. and Thornton, E. A., The Finite Element Method for Engineers, Second Edition, Wiley, 1982.

8. Stroud, A. H. and Secrest, D., Gaussian Quadrature Formulas, Prentice-Hall, Inc., 1966.

9. Sokolnikoff, I. S. Mathematical Theory of Elasticity, Second Edition, R. E. Krieger Publishing Co., 1983.

APPENDICES

APPENDIX ~~A~~

8 Node, Quadratic Isoparametric Plane Stress (Strain) Element

General Formulation

The element used in Finel is based on the displacement method where the form of the displacement within the body (element) is assumed. The variational principal used is the principle of minimum potential energy.

The potential energy functional [7] can be written as :

$$
\begin{aligned}
(u,v) = \frac{1}{2} & \int_A \{\delta\}^T [B]^T [C] [B] \{\delta\} t\,dA \\
- & \int_A \{\delta\}^T [B]^T [C] \{\epsilon o\} t\,dA \\
- & \int_A \{F\}^T \{\delta\} t\,dA - \int_S \{T\}^T \{\delta\} ds
\end{aligned}
\qquad \text{(A-1)}
$$

where:

t = t(x,y) = thickness of the panel (plane stress only).

$\{\delta\} = \begin{matrix} u(x,y) \\ v(x,y) \end{matrix}$ = components of displacement field.

$$[B] = \begin{bmatrix} \frac{\partial}{\partial x} & 0 \\ 0 & \frac{\partial}{\partial y} \\ \frac{\partial}{\partial y} & \frac{\partial}{\partial x} \end{bmatrix}$$ = matrix relating strains and displacements.

[C] = material stiffness matrix, which takes different forms depending on the type of problem.

$\{\epsilon_0\}$ = column vector of initial strains.

$\{F\} = \begin{Bmatrix} X \\ Y \end{Bmatrix}$ = body force components.

$\{T\} = \begin{Bmatrix} T_x \\ T_y \end{Bmatrix}$ = boundary traction components

The two dimensional problems which can be modelled by Finel are plane stress and plane strain in a panel of isotropic material with constant material properties throughout each element. For these problems the material stiffness matrices are:

Plain Stress

$$[C] = \frac{E}{(1 - \nu^2)} \begin{bmatrix} 1 & \nu & 0 \\ \nu & 1 & 0 \\ 0 & 0 & \frac{1-\nu}{2} \end{bmatrix}$$

Plane Strain

$$[C] = \frac{E}{(1 - \nu)(1 - 2\nu)} \begin{bmatrix} 1-\nu & \nu & 0 \\ \nu & 1-\nu & 0 \\ 0 & 0 & \frac{1-2\nu}{2} \end{bmatrix}$$

Program Finel does not model initial strains nor body force components. Thus only the first and last terms of equation A-1 are considered. At equilibrium, the potential energy is minimized, therefore:

$$\begin{Bmatrix} \frac{\partial \pi}{\partial u} \\ \\ \frac{\partial \pi}{\partial v} \end{Bmatrix} = \{0\} = {}_A\iint [B]^T [C] \; [B] \; \{\tilde{\delta}\} \, tdA \\ - {}_S\int \{T\} \, dS$$

Interpolation Functions

The true displacement field is not known, so some displacement function must be assumed. The usual procedure is to assume that the displacement within the body is described by a polynomial. This requires that the displacements must be known at as many locations in the body as there are coefficients in the polynomial. The simplest polynomial possible which fulfills the compatibility and completeness conditions is the linear polynomial:

$$\Psi(x,y) = ax + by + c$$

Note that the complete linear polynomial has been truncated by the 'xy' term. For each component of the displacement field:

$$u(x,y) = ax + by + c$$

$$v(x,y) = dx + ey + f$$

So for each function, u(x,y) and v(x,y), the value must be known at three locations (referred to as nodes) in the body in order for the sets of coefficients a, b, c, and d, e, f to be evaluated. This is accomplished with a triangular element, the displacements being described at each of the three vertices.

The displacements at the nodes, in terms of the polynomial are:

$$u_1(x_1,y_1) = ax_1 + by_1 + c$$
$$u_2(x_2,y_2) = ax_2 + by_2 + c$$
$$u_3(x_3,y_3) = ax_3 + by_3 + c$$

$$v_1(x_1,y_1) = ax_1 + by_1 + c$$
$$v_2(x_2,y_2) = ax_2 + by_2 + c$$
$$v_3(x_3,y_3) = ax_3 + by_3 + c$$

Written in matrix form:

$$\{u\} = [g_i]\{A_1\}$$
$$\{v\} = [g_i]\{A_2\}$$

where

$\{u\}$, $\{v\}$ = the nodal displacements

$\{A_1\}$, $\{A_2\}$ = column vectors of polynomial coefficients

$[g_i]$ = matrix of polynomial terms, each row containing terms evaluated at the coordinates of a node.

The coefficient vectors can now be solved for and used in the description of the displacement field everywhere within the element:

$$\{A_1\} = [g_i]^{-1}\{u\}$$
$$\{A_2\} = [g_i]^{-1}\{v\}$$

$$u(x,y) = \{g(x,y)\}^T\{A_1\}$$
$$v(x,y) = \{g(x,y)\}^T\{A_2\}$$

$$u(x,y) = \{g(x,y)\}^T[g_i]^{-1}\{u\}$$
$$v(x,y) = \{g(x,y)\}^T[g_i]^{-1}\{v\}$$

$$u(x,y) = \{L(x,y)\}^T\{u\}$$
$$v(x,y) = \{L(x,y)\}^T\{v\}$$

Here $\{L\}$ contains the interpolation functions.

The partial derivatives of the interpolated displacement field can also be described once the coefficients have been evaluated.

This notation is valid for any order polynomial and so applies to higher order elements as well as the linear triangular element. The element used in Finel has 8 nodes and uses a quadratic interpolation polynomial truncated by the x^2y^2 term.

Now that the displacement field and its partial derivatives can be expressed in terms of the displacements at the nodes, they can be substituted into (A-1) to produce a set of linear equations.

$$\int_A [B^e]^T [C][B^e] t\,dA \;\{\delta^e\} = \int_S \{T^e\} ds$$

or:

$$[K][\delta] = \{F\}$$

where:

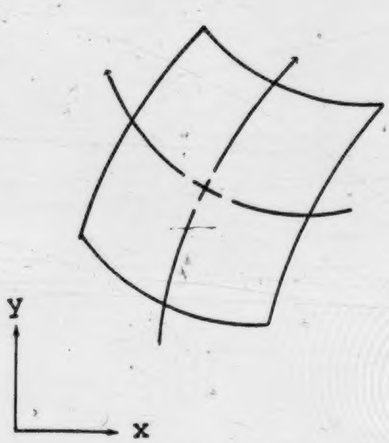
$$[B^e] = \begin{bmatrix} \frac{\partial \{L\}^T}{\partial x} & 0 \\ 0 & \frac{\partial \{L\}^T}{\partial y} \\ \frac{\partial \{L\}^T}{\partial y} & \frac{\partial \{L\}^T}{\partial x} \end{bmatrix}$$

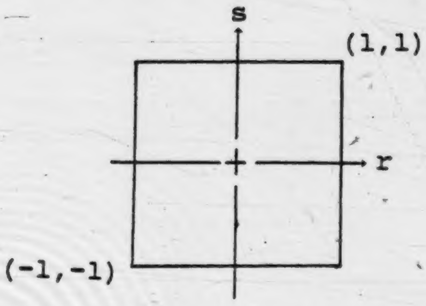
This system can then be solved for the nodal displacements and hence an approximation of the displacement and stress field.

Isoparametric Transformation and Numerical Integration.

Integration of the set of equations over the element volume can often be difficult, if not impossible, to perform analytically. For the linear triangular element the linear displacement function produces a constant stress, constant strain element which makes integration of the stiffness matrix and load vectors straightforward. However, higher order elements, with their curved boundaries and nonconstant stress fields make integration considerably more difficult. The term 'volume' is a general term referring to a region in n-dimensional space. For plane problems, the two dimensional volume refers to the area.

Isoparametric transformation maps the global element space into a simple local space, greatly simplifying the integration procedure. It is the most common type of coordinate transformation which uses 'shape functions' to map the global system to the local system. Isoparametric transformation uses the same interpolation functions as those used for the displacement field to evaluate the global coordinates corresponding to the location of a point in the local coordinate system:





$$u(r,s) = \{L(r,s)\}^T\{u\}$$
$$v(r,s) = \{L(r,s)\}^T\{v\}$$

$$x(r,s) = \{L(r,s)\}^T\{x\}$$
$$y(r,s) = \{L(r,s)\}^T\{y\}$$

In contrast, sub-parametric and super-parametric transformations use lower and higher order interpolation functions respectively. The disadvantage of each of these is that different nodal systems must be used for displacements and coordinates.

Because every element is mapped into the same local coordinate space, the interpolation functions do not change from one element to another and so need not be re-evaluated for each element matrix generation.

However, two additional identities must be introduced to distinguish one element from another:

$$\begin{Bmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \end{Bmatrix} = [J]^{-1} \begin{Bmatrix} \frac{\partial}{\partial r} \\ \frac{\partial}{\partial s} \end{Bmatrix}$$

and

$$dxdy = |J|drds$$

where

$$[J] = \begin{bmatrix} \frac{\partial x}{\partial r} & \frac{\partial y}{\partial r} \\ \frac{\partial x}{\partial s} & \frac{\partial y}{\partial s} \end{bmatrix} = \begin{bmatrix} \frac{\partial \{L\}^T}{\partial r} \{x\} & \frac{\partial \{L\}^T}{\partial r} \{y\} \\ \frac{\partial \{L\}^T}{\partial s} \{x\} & \frac{\partial \{L\}^T}{\partial s} \{y\} \end{bmatrix}$$

Using these identities, the function (or functional) can be mapped into the local area and integrated easily. This simple local region is ideally suited to numerical integration by Gaussian Quadrature [8]. Integration over the area is expressed as:

$$\int_{-1}^{1}\int_{-1}^{1} f(r,s)drds = \sum_{i=1}^{NG}\sum_{j=1}^{NG} W_iW_jf(r_i,s_j)$$

, where NG is the number of gauss points in each dimension.

Applying this to (2), the matrix product is evaluated at each Gauss point, multiplied by the weighting factors and summed together to produce the element stiffness matrix.

$$[K] = \int_A [B^e]^T[C][B^e]tdxdy$$

$$[K] = \int_{A'} [B^e]^T[C][B^e]t|J|drds$$

$$[K] = \sum_{i=1}^{NG}\sum_{j=1}^{NG} [B^e]^T[C][B^e]t|J|drds$$

where

$$[B^e] = \begin{bmatrix} \frac{\partial\{L(x,y)\}^T}{\partial x} & 0 \\ 0 & \frac{\partial\{L(x,y)\}^T}{\partial y} \\ \frac{\partial\{L(x,y)\}^T}{\partial y} & \frac{\partial\{L(x,y)\}^T}{\partial x} \end{bmatrix}$$

and

$$\begin{Bmatrix} \frac{\partial\{L(x,y)\}^T}{\partial x} \\ \frac{\partial\{L(x,y)\}^T}{\partial y} \end{Bmatrix} = [J^{-1}] \begin{Bmatrix} \frac{\partial\{L(r,s)\}^T}{\partial r} \\ \frac{\partial\{L(r,s)\}^T}{\partial s} \end{Bmatrix}$$

APPENDIX B

GLOBAL SYSTEM FORMATION

Matrix Structure Formation

The sparse storage technique used by Finel requires that the structure of the matrix be determined before the elements can be assembled. This is accomplished by searching all elements for each unconstrained degree of freedom and allocating a space in the row for each coupled unconstrained degree of freedom numbered larger. Before beginning the matrix construction search, the row pointer vector is initialized to indicate partitioned rows and columns:

pointer(i) = 0; displacement is constrained.
pointer(i) = 1; displacement is unconstrained.

After allocating space to each row it may or may not be desirable to sort the column numbers in the row. If the matrix is densely populated and considerable random access to the matrix is necessary, a binary search may improve performance considerably. However, for the two dimensional elastic problem the matrix is not very dense and need be assembled only once, so a linear search was used eliminating the need for a column number sort.

Matrix Assembly

To produce the system of linear equations, eight sets of information are needed:

1) Coordinates of the nodes.
2) Panel thicknesses at the nodes (for plane stress).
3) Global node numbers corresponding to element node numbers.
4) Element material properties.
5) Constrained nodes (boundary conditions).
6) Model loading information.
7) Dependent nodal displacements.
8) Skewed coordinate systems for some nodes.

While items 1 through 6 are straight forward 7 and 8 deserve a more detailed explanation. To model a frictionless interface between two boundaries, normal displacements of two contacting surfaces are constrained to be equal. This is accomplished by defining an array of pointer pairs specifying the dependent displacements. One degree of freedom is constrained and all element entries associated with it are redirected to the other, now independent, degree of freedom. This information must be considered when the matrix structure is being formed.

The displacements of most nodes are usually determined by their x and y components. For some of the boundary nodes, however, it is desirable to express the displacements in terms of another coordinate system which has been rotated through some angle, usually to give displacement components normal and tangent to the boundary. For this reason, a skewed coordinate system can be defined for some nodes by specifying an angle of rotation. This allows the model to be constrained in directions other than vertical and horizontal.

The first step is forming the matrix structure as defined in the previous subsection. After defining the partitioned rows and columns, the element node numbers and dependent variable numbers are loaded into main memory. Then, for each degree of freedom, each set of eight element node numbers is converted to 16 displacement numbers, dependent displacements are replaced with independent counterparts, and space is allocated for each coupled variable.

Once the system structure has been defined, all element node numbers are no longer required in main memory and are flushed. In their place, the nodal coordinates and panel thicknesses are retrieved from disk and element matrix generation can begin.

The node numbers and material properties are read from disk for each element in turn. Using the coordinates and panel thicknesses corresponding to the node numbers and the element properties, the element stiffness matrix can be formed. Next, the eight element node numbers are converted to 16 dependent/independent variable numbers and all dependent variables are replaced with their independent counterparts. Before the element matrix can be assembled into the global matrix each node must be checked to determine if a skewed coordinate system is used and, if so, the matrix is transformed accordingly. Transformation must be done before assembly because the matrix structure does not provide room for partitioned rows and columns making constrained nodes impossible to transform.

The final stage in producing the system of equations is generating the forcing vector. The loads can include point loads and equivalent loads due to tractions applied to the boundaries. No provision was made for body forces as they were not a consideration in the problem at hand, but they could easily be incorporated into the loading of the model. Point load components are simply added to the corresponding rows of the load vector while tractions are integrated isoparametrically over each loaded element boundary to produce a set of three equivalent loads which are assembled into the load vector. Once all loads have been entered, forces corresponding to dependent variables are redirected to the independent variables and the load vector is transformed for those nodes using a skewed coordinate system.

APPENDIX C

System Solver

The form of the system to be solved is:

$$[A]\{X\} = \{F\}$$

or:

$$\sum_{j=1}^{n} a_{ij}x_j = f_i$$

If the i'th equation is written in terms of x_i:

$$x_i = \frac{1}{a_{ii}} \quad f_i - \sum_{\substack{j=1 \\ j\neq i}}^{n} a_{ij}x_j$$

This produces n expressions, one for each x_i, in terms of f_i and x_j, $j = 1..n$, $j = i$. Each iteration of the solution vector, {X} will produce a better approximation to the true solution if the matrix [A] is diagonally dominant, which will always be true for a properly constrained finite element stiffness matrix.

In Gauss-Siedel iteration the most recent information is used in the iteration expression. For example, suppose the k'th iteration (denoted by superscript k) is complete and iteration k+1 is being executed. In the calculation of x_i, the algorithm can use x_j from iteration k+1 for $j < i$ since they have already been evaluated. The iteration formula now becomes:

$$x_i^{k+1} = \left[\frac{1}{a_{ii}} \; f_i - \sum_{j=1}^{i-1} a_{ij} x_j^{k+1} - \sum_{j=i+1}^{n} a_{ij} x_j^{k}\right] \qquad \text{(C-1)}$$

where k is the number of the most recently completed iteration.

Instead of using this value for x_i, it may desirable to interpolate (under-relax) or extrapolate (over-relax) the newest value. Incorporating a relaxation factor, w, the iteration formula becomes:

$$x_i^{k+1} = x_i^{k}(1-w) + \frac{w}{a_{ii}} \left[f_i - \sum_{j=1}^{i-1} a_{ij} x_j^{k+1} - \sum_{j=i+1}^{n} a_{ij} x_j^{k}\right]$$

A unity relaxation factor results in no relaxation and the expression becomes equivalent to (1). If the system of equations is unstable, under-relaxation may be necessary and w less than one may be required for convergence. For stable systems such as those used in the finite element method over-relaxation can be used to greatly enhance the convergence process. In this case w may be greater than one but must be less than two, though oscillations in the solution can occur if w is too close to two. For most systems a suitable relaxation factor is in the range $1.8 < w < 1.95$.

The program Finel, because of its storage technique, implements this Gauss-Siedel algorithm in a unique way. An additional vector is used to store partial sums for products from the unstored lower triangular half of the matrix, eliminating an expensive search for matrix entries to the left of the diagonal entry from the upper triangular values for each row.

For each x_i it is assumed that the first summation term has already been evaluated during the evaluation of x_j, $j < i$ and is stored in the extra vector space. The second summation term is easily evaluated by summing the products of the matrix entries in the row with the l'th components of the vector $\{x\}^k$, where l is the column number stored with the value. Now the new value for x_i can be calculated and is placed in the extra vector space.

This new value for x_i will be used in the calculation of some of the subsequent x_j's. Since the matrix is symmetric, the column beneath the diagonal is the same as the row to its right, and it is those rows which have non-zero entries in this column which are dependent upon this new x_i. Once again, the program moves across the row, this time adding the product of the matrix entry and x_i to the first summation term for the row l, where l is the column number stored with the matrix entry.

Because there is a unique combination of all the products of the stiffness matrix entries with the solution vector, and because the stiffness matrix is symmetric, the correct evaluation of the first summation term for x_i is assured by the time row i is reached by the iteration procedure.

Once the iteration is complete the two solution vectors contain the last solution approximation and the latest approximation. This gives the program an opportunity to test for convergence of the process, or to pass the convergence test which results in fewer numerical operations and improved performance. In Finel the convergence test is user selectable and can be toggled during the solution process.

Before the next iteration can begin, the pointers to $\{x\}^k$ and $\{x\}^{k+1}$ are swapped and storage for the old vector is initialized to zero for use as the partial sums. The procedure will be repeated until suitable convergence is attained.

By combining the matrix storage described earlier with this solution algorithm, the number of operations required per iteration is reduced to the absolute minimum since all entries containing zero are removed entirely from the process. This technique would obviously be well suited to programs which use auxiliary storage to contain the stiffness matrix because of the reduced disk space required and the resulting decrease in data transfers from disk to memory that would be necessary for a given problem.

APPENDIX D

Program Listing for

FINEL

```
/*

FILE : FINEL.C */
/*=====================================================================

                Finite Element Program
            Two Dimensional Stress Analysis
              Plane Stress - Plane Strain

   Input is in six files:
       1) file.cd -    This file contains coordinates for each node.
               format:
                       number_of_nodes
                       node    x_coordinate    y_coordinate
                       ...      ...             ...
       2) file.thN -   contains thicknesses for each of the overlays,
                       with the overlay number denoted by N.  The overlay
                       numbers must be sequential and start at 1.
                       Only changing thicknesses need be entered, i.e.
                       the thickness of on node is read and is assigned
                       to that node and each consecutive node thereafter
                       up to the next node read.  This means that the nodes
                       must be in numerical order.
               format:
                       node    thickness
                       ...      ...

       3) file.elN  -  Each line contains the element number
                       followed by eight integers, each
                       corresponding to the respective global
                       nodes for each of the element nodes.
                       Nodes are numbered starting at one corner
                       of the element and going around the element
                       in a counterclockwise direction.
                           The modulus of elasticity and poissons
                       ratio are both initialized to zero.  Their
                       values are changed by appending them to the
                       end of the element node numbers on the same row
                       and are maintained until changed on a subsequent
                       row.
               format:
                           node1   node2   node3 ...   node8   [E    nu]
                            ...     ...     ...         ...

       4) file.pt -    This file contains the number of nodes in the
                       system as well as the constrained nodes.
                       The first line contains the number of nodes
                       and the number of constrained nodes.  Following
                       that, each line contains an integer
                       for the node constrained and a one
                       or two character string.  The string
                       contains one x for constraint in x
                       direction and one y for constraint
```

```
                   in y direction.
          format:
                   number_of_constrained_nodes
                   node       xy
                   node       yx
                   node       x
                   node       y
                   node       X
                   ...        ...

      5) file.trN -  This contains the boundary  'tractions' applied
                   to the system for the Nth overlay.  There MUST be
                   a traction file for each overlay even if it is just
                   one line.  Each line contains a node number and the
                   x and y tractions applied at that node.
          format:
                   node    x_traction      y_traction
                   ...        ...              ...

      6) file.ldN -  This file contains triplets of nodes for each
                   element that has a boundary with a traction
                   applied.  This allows a parabolic distribution
                   of the traction along the element boundary.
                   Each row contains the three boundary nodes for the
                   boundary element.
          format:
                   node1      node2      node3
                   ...        ...        ...
==================================================================*/

#include "math.h"
#include "stdio.h"
#include "graphics.h"
#include "mylib.h"
extern int _oldfile;
/*---------------------------------------------------------------
                  EXTERNAL VARIABLES
-----------------------------------------------------------------*/

double N(), jacobinv();                /* N() is the interpolation fn */
char istress;                        /* flag for plane stress or strain  */
char file[100];                        /* I/O file names             */
char filenum[2];                     /* number for overlays        */
float *nodecord;
double CC[3][3] = {                    /* elasticity matrix - modified */
        { 1.0 , 1.0 , 0.0 },         /* later depending on whether   */
        { 1.0 , 1.0 , 0.0 },         /* plane stress or strain       */
        { 0.0 , 0.0 , 1.0 }
   };

extern unsigned matrix, diags;       /* paragraphs for stiffness     */
                                     /* matrix and index arrays      */
                                     /* use to form stress array     */

/*================================================================
                   ******************
                   *  MAIN PROGRAM  *
                   ******************                         */

main()
{
/*------------------------------------------------------------------
                        DEFINE VARIABLES
----------------------------------------------------------------*/

extern char file[];
extern char filenum[2];
unsigned dim;                     /* dim is number of NODES */
unsigned formt();
double modulus, poisson;          /* material constants */
float *formcord();                /* coordinate array and routine */
float *thick;
float *displace;                  /* pointer to displacement vector */

/*------------------------------------------------------------------
                           SET UP SYSTEM
------------------------------------------------------------------*/

   freeall(3072);                    /* leave a large stack space */
   scr_setup();
   scr_clr();
   scr_color(0, 1);                  /* set blue border          */

/*------------------------------------------------------------------
               READ DATA and FORM SYSTEM OF EQUATIONS
------------------------------------------------------------------*/

/* get name of files for input/output; also gets number of int. points */
   getname();
   strcpy(filenum, "1");

/* get coordinates of all the nodes and the size of the system       */
   nodecord = formcord();          /*  returns pointer to cord array    */
   dim = *nodecord;                /*  first element has size of system */

/* initialize global matrix */
   initialize(dim*2);

/* get boundary information   */
   partition();                    /* partition constrained nodes       */
   formredir();                    /* get redirection data              */
   formtrans();                    /* get angles for boundary transforms */

/* prepare stiffness matrix            */
   formstiff(dim);

/* make room for force and displacement vectors - always in use */
   if (!(displace = calloc(2 * (dim+1), sizeof(float)))){
      printf("\n No room for displacement vectors");
      exitl(0);
   }

/* make room for thickness vector   */
   if (!(thick = calloc(dim+1, sizeof(float)))){
       printf("\n No room for thickness vector");
       exitl(0);
   }


/*-------------------------------------------------------------------*/
/* for each overlay, get thickness vector and add matrices to global */

while(formt(thick, dim)){          /* gets thickness array */
                                   /* or returns zero if no file */


/* add force vector for this overlay    */
       load(displace, thick, dim);
/* form global stiffness matrix */
       formglob(nodecord, thick, dim);

/* increment overlay pointer */
       filenum[0]++;
}
/*-------------------------------------------------------------------*/
/* add nodal point loads to load vector */
   pload(displace);

/* transform load vector */
   loadtrans(displace);

/* thickness vector not needed now; make room for solver workspace */
   free(nodecord);
   free(thick);

/*--------------------------------------------------------------
           SOLVE FOR DISPLACEMENTS AND STRESSES
-------------------------------------------------------------*/
/* solve for displacements */
   solve(displace, 2 * dim);       /* solver is bound in and can */
   scr_clr();                      /* be direct or itterative */

/* adjust displacements for redirection    */
   unredir(displace);

/* transform displacements back    */
   disptrans(displace);

/*-------------------------------------------------------------------


/* empty stiffness matrix to hold stress data  */
unsigned i;
for (i = 0; i < dim * 4; i++)
   stored(matrix, i, 0.0);
```

```
nodecord = formcord();       /*  returns pointer to cord array   */
/* for each overlay form stress report.     -    */
   strcpy(filenum, "1");
while(formt(0, dim)){             /* or returns zero if no file */
/* generate stress vectors */
   formstrs(nodecord, displace, dim);

/* increment overlay pointer */
       filenum[0]++;
}

/* print stress vectors          */
   prntstrs(dim, displace);

/*------------------------------------------------------------------------*/


/* end program */
   exit(0);
}

/*========================================================================

                 ************************************
                 *  FORM MATERIAL STIFFNESS TENSOR  *
                 *            formCC(E, nu)         *
                 ************************************              */

formCC(modulus, poisson)
double modulus, poisson;
{
extern double CC[3][3];
extern char istress;
double const, a;


/* type is determined by istress - stress or strain */
/* now generate [CC]                         */
   CC[0][2] = CC[1][2] = CC[2][0] = CC[2][1] = 0.0;

   if (istress == '1') {            /* plane stress */
       const = modulus / (1.0 - poisson * poisson);
       CC[0][0] = CC[1][1] = const;
       CC[0][1] = CC[1][0] = poisson * const;
       CC[2][2] = const * (1.0 - poisson) / 2.0;
   } else {
       const = modulus / ((1.0 + poisson) * (1.0 - 2.0 * poisson));
       CC[0][0] = CC[1][1] = const * (1.0 - poisson);
       CC[0][1] = CC[1][0] = poisson * const;
       CC[2][2] = const * 0.5 * (1.0 - 2.0 * poisson);
   }
   return;
}

/*======================================================================
               ***************************************
               *  READ COORDINATES & RETURN POINTER  *
               *       coords = formcord()           *
               *  data format:                       *
               *  number_of_nodes                    *
               *  node   x_coord  y_coord            *
               ***************************************          */

float *formcord()          /* reads coordinates of nodes    */
{
extern char file[];                /* data file name             */
char cordfile[100];
unsigned i, j, k;               /* counters                   */
int input;                      /* file specifier             */
float *nodecord;                /* pointer to coordinate array */
unsigned node, dim;                /* node number read from file  */
double x, y;                    /* x, y coordinate read from file */

   strcpy(cordfile, file);      /* form name of input file    */
   strcat(cordfile, ".cd");
   scr_clr();
   scr_rowcol(10, 20);
   dim = 0;
/*--------------------------------------------------------------
OPEN INPUT FILE - error checking included  */

   input = open(cordfile, 3);
   if (input < 1){
       printf("Cannot open coordinate file.");
       exit1(0);
   }
   printf("Reading coordinate file: %s \n", cordfile);
/*--------------------------------------------------------------
READ DATA                                                    */

   fscanf(input, "%u", &node);
   if(!node){
       printf("\nError in coordinate file");
       exit1(0);
   }
   dim = node;
   scr_rowcol(12, 20);
   printf("Number of nodes = %u", dim);

/* allocate space for the array - allocate extra space for use in solver   */
   if (!(nodecord = calloc((dim + 1)*2, sizeof(float)))){
       printf("\n No room for coordinate array");
       exit1(0);
   }
   *nodecord = dim;                            /* store dimension of system    */

/* read coordinates */
   for (i = 1; i <= dim; i++){
       fscanf(input, "%u%F%F", &node, &x, &y);
       if (node > dim){
```

```
      printf("\nNode larger than system size - line %u", i);
      exitl(0);
    }
    node = 2 * (node);
    *(nodecord + node) = x;
    *(nodecord + node + 1) = y;
  }

  close(input);
_oldfile = 99;
  return(nodecord);

}
/*════════════════════════════════════════════════════════════
      ********************************************
      *  READ NODAL PLATE THICKNESSES            *
      *  error = formt(thick, dim)               *
      *                                          *
      *  data format:                            *
      *  node   thickness                        *
      *  NOTE: only varying thicknesses need be  *
      *        entered; for constant thickness   *
      *        enter only one nodal value        *
      ********************************************        */


unsigned formt(thick, dim) /* reads material constants       */
float *thick;              /* pointer to coordinate array    */
unsigned dim;                /* number of nodes                */
{
extern char file[];
extern char filenum[];
char tfile[100];
unsigned i, j;
int k, input;
double thck;

  strcpy(tfile, file);         /* get name of input file */
  strcat(tfile, ".th");
  strcat(tfile, filenum);      /* add overlay number    */

  scr_clr();
  scr_rowcol(10, 20);

  input = open(tfile, 0);
  if(input < 1)                /* if no such overlay        */
      return(0);               /* return no overlay flag    */
  if(!(thick)){                /* if no thickness vector    */
      close(input);            /* close the file            */
_oldfile = 99;
      return(1);               /* return overlay flag       */
  }

  scr_clr();
  scr_rowcol(10, 20);
```

```
    for (i = 0; i <= dim; i++)      /* empty thick  */
        *(thick + i) = 0;

    while (fscanf(input, "%u%F", &j, &thck) > 0)    /* read thicknesses */
        *(thick+j) = thck;

    j = 0;
    while(!*(thick + j) && (j <= dim))  /* seek to first nonzero t from start
        j++;

    if (j > dim){
        printf("\nNo Thickness in range");
        ci();
        return(0);
    }

    for (i = 0; i <= dim; i++){     /* fill in the rest of thick    */
        if(*(thick + i))
            j = i;
        else
            *(thick+i) = *(thick+j);
    }

    close(input);
_oldfile = 99;
    return(1);
}

/*==================================================================
               ************************************
               *   FORM GLOBAL STIFFNESS MATRIX  *
               *   formglob(coords, thick, dim)  *
               ************************************          */

formglob(coord, thick, dim)         /* forms glogal stiffness matrix and retur
                                number of elements          */
float *coord, *thick;       /* x[dim*2, 3] is coordinate array  */
unsigned dim;
{
extern char file[];
extern char filenum[];
char elfile[100];
extern double CC[][];           /* material stiffness matrix        */
unsigned i, ii, j, elnumber;    /* elnumber is counter for elements */
unsigned *elnode;               /* pointer to array with elmnt node #'s */
int flag, nscanf;               /* flag and number of items read    */
int k, input;
double *elstiff, modulus, poisson;
double templ, temp2;
double elstiff[136];
unsigned elnode[16];
/*------------------------------------------------------------------
OPEN-INPUT FILE - check for errors                                 */
```

```
    strcpy(elfile, file);
    strcat(elfile, ".el");
    strcat(elfile, filenum);        /* add overlay number   */

    scr_rowcol(10,20);
    input = open(elfile, 3);
    if (input < 0){
        printf("Cannot find element data file %c", elfile);
        exitl(0);
    }

/*-----------------------------------------------------------------------
FORM STIFFNESS MATRICES and ADD TO GLOBAL MATRIX                        */

    initgraph(coord, dim);
    scrn(1);
    fscanf(input, "%u", &elnumber);
    for (ii = 0; ii < elnumber; ii++){
        nscanf = fscanf(input, "%u%u%u%u%u%u%u%u%u%F%F", &flag, elnode, elnode
        elnode+2, elnode+3, elnode+4, elnode+5, elnode+6, elnode+7,
        &modulus, &poisson);
        if (nscanf < 0){
            printf("\nFewer elements in data file than specified");
            break;              /* at end exit loop */
        }
        if (nscanf < 9){
            printf ("\nError in element data file, element #%d", ii);
            exitl(0);
        }

        if (nscanf > 10){           /* if there is a new modulus    */
            formCC(modulus, poisson);
            scr_rowcol(24,0);
            printf("E = %-7.0F  Nu = %-5.2F ", modulus, poisson);
        }
        scr_rowcol(0, 34);
        printf("%d", ii+1);
        eldraw(0, coord, elnode);
/* form element stiffness matrix */
        isoparm(coord, thick, elnode, CC, elstiff);
/* adjust elnode for assemble */
        for(i = 0,j = 7; i < 8; i++,j--){
            *(elnode + 2*j+1) = *(elnode + j) * 2;
            *(elnode + 2*j) = *(elnode + j) * 2 - 1;
        }
/* adjust elnode for transform and assemble    */
        rednode(elnode);
/* transform element matrix if necessary    */
        eltrans(elstiff, elnode);
/* assemble elememt matrix into global matrix  */
        assemble(elstiff, elnode, 16);
        if(toupper(csts()) == 'E'){         /* if there is an error */
            exitl(0);
        }
    }
```

```
   scrn(0);
   close(input);
_oldfile = 99;

   scr_clr();
   return;
}

/*==============================================================
               ***************************************
               *   PARTITION STIFFNESS MATRIX        *
               ***************************************/

partition()                         /*  partitions stiffness matrix */
{
unsigned i, j, k, n, nconst;
extern unsigned diags;          /* paragraph to diagonals array */
int intcmp(), input;            /* intcmp() compares integers   */
extern char file[];                 /* for qsort()                  */
char partfile[100];           /* name of partition file       */
char xy[5];                   /* string that tells if x and/or */
                                /* y partitioned                */
   strcpy(partfile, file);
   strcat(partfile, ".pt");

   input = open(partfile, 3);
   while (input == ERR){
       scr_clr();
       scr_rowcol(10,20);
       printf("Cannot find constraint file - %s. Enter file name: ", partfile
       scanf("%s",partfile);
       input = open(partfile, 3);
   }
   scr_clr();
   scr_rowcol(10,20);

   printf("\nReading constraint data file: %s\n", partfile);

/* get element data */
   fscanf(input,"%u", &nconst);    /*  read number of constrained nodes */

   k = 1;
   for (i = 0; i < nconst; i++){
       fscanf(input, "%u%s", &n, xy); /* read node and direction  */
       n = 2*n - 1;                  /* row number of stiffness matrix   */
       switch (toupper(xy[0])){      /* see if constrained in x &/or y */
           case 'X':                 /* direction(s) */
           case 'F':                 /* or first for transformed */
               storeu(diags, n, 0);
               break;
           case 'Y':                 /* for y constraint or      */
           case 'S':                 /* second node in element   */
               storeu(diags, n+1, 0);
               break;
           case 'B':                 /* if both displacements at node    */
```

```
              storeu(diags, n, 0);
              storeu(diags, n+1, 0);
              break;
       }
   }
   close(input);
_oldfile = 99;

   scr_clr();
   return;
}
/*==============================================================
              *****************************************
              *   FORM ELEMENT_STRESSES and REPORT    *
              *   FOR A GIVEN OVERLAY                 *
              * formstrs(coord, disp, dim)    *
              *****************************************/

formstrs(coord, disp, dim)
float *disp;            /* displacement vector */
float *coord;
unsigned dim;

{
extern char file[];
extern char filenum[];
extern int nint, nintt;
extern unsigned matrix;                     /* stiffness matrix and index   */
extern double ntc[][2];
extern double CC[3][3];
int input, t, flag, nscanf;
unsigned i, ii, j, k, m, p, q, nel;
unsigned sigx, sigy, sigxy, sign;      /* paras for x, y, shear stress */
double const, modulus, poisson;
double L, xx, yy;
double *stresst, *st, *L1d, *Ld;
double sig[3];
char elfile[100];
static char extin[4] = ".el";          /* input file extension  */
unsigned elnode[8];
double epsilon[2][2];  /* derivatives wrt x top row */
int inode[8][2] = {
   { 1,  1},
   { 1,  0},
   { 1, -1},
   { 0, -1},
   {-1, -1},
   {-1,  0},
   {-1,  1},
   { 0,  1}
};
double L1d[10];
double stresst[75];
/*---------------------------------------------------------------
* GET WORKING VECTORS                                          */
```

```
   sigx = matrix;                  /* paragraphs for stress vectors    */
   sigy = matrix + (dim+6) / 2;
   sigxy = matrix + 2 * ((dim+6) / 2);
   sign = matrix + 3 * ((dim+6) / 2);

/*----------------------------------------------------------------------
PREPARE FILES FOR I/O                                                 */
   strcpy(elfile, file);
   strcat(elfile, extin);
   strcat(elfile, filenum);     /* add overlay extension    */
   input = open(elfile, 3);
   while (input == ERR){
       printf("Cannot find element data file - %s. Enter file name: ", elfile
       scanf("%s",elfile);
       input = open(elfile, 3);
   }

   scr_clr();
/*------------------------------------------------------------------------
READ ELEMENT DATA AND PRODUCE STRESS RESULTS                             */

   ii = 0;
   fscanf(input, "%u", &nel);
   scr_rowcol(10, 1);
   printf("element:");
   for (ii = 0; ii < nel; ii++){
       nscanf = fscanf(input, "%u%u%u%u%u%u%u%u%u%F%F", &flag, elnode, elnode
       elnode+2, elnode+3, elnode+4, elnode+5, elnode+6, elnode+7,
       &modulus, &poisson);
       if (nscanf < 1){
           printf("\nFewer elements than specified in element data file.");
           break;              /* at end exit loop */
       }
       if (nscanf < 9){
           printf("\nError in element data file; element #%d", ii);
           exit(0);
       }
       if (nscanf > 10){            /* if there is a new modulus     */
           formCC(modulus, poisson);
           scr_rowcol(24,0);
           printf("E = %-7.0F  Nu = %-5.2F ", modulus, poisson);
       }
       scr_rowcol(10,10);
       printf("%-4d\n", ii+1);
       scr_clrl();

       st = stresst;                          /* start of stress matrix  */
       for (i = 0; i < nint; i++){            /* form strain matrix   */
           for( j = 0; j < nint; j++){        /* at each gauss point  */
               formeps(disp, elnode, coord, ntc[nintt + i][0],
               ntc[nintt + j][0], epsilon);
       /* form strain matrix  */

/* calculate stresses at the gauss point */
               *(st    ) = CC[0][0] * epsilon[0][0] + CC[0][1] * epsilon[1][1
```

```
        *(st + 1) = CC[1][0] * epsilon[0][0] + CC[1][1] * epsilon[1][1
        *(st + 2) = CC[2][2] * (epsilon[0][1] + epsilon[1][0]);
        st += 3;
      }
    }       /* stresses have been calculated at each gauss point    */
/* get stresses at the nodes and average in with the global node values    */
    for (i = 0; i < 8; i++){                /* for each node   */
      for (j = 0; j < nint; j++){      /* form 1-D Lagr. interp.   */
        Ld = (Lld + 2 * j);
        *(Ld)  = *(Ld + 1) = 1.0;
        for (k = 0; k < nint; k++){
          if (k != j){
            *(Ld    ) *= (ntc[nintt + k][0] - inode[i][0]) /
/* x interpolation */         (ntc[nintt + k][0] - ntc[nintt + j][0]);
            *(Ld + 1) *= (ntc[nintt + k][0] - inode[i][1]) /
/* y interpolation */         (ntc[nintt + k][0] - ntc[nintt + j][0]);
          }
        }
      }       /* interpolation values have been calculated for node   */

      st = stresst;
      sig[0] = sig[1] = sig[2] = 0.0;     /* x, y, and shear stress   */
      for (j = 0; j < nint; j++){         /* for the node             */
        for (k = 0; k < nint; k++){       /* calc stresses @ node     */
          L = *(Lld + 2 * j) * *(Lld + 2 * k + 1);
          for (m = 0; m < 3; m++){
            sig[m] += *st * L;
            st++;
          }
        }
      }          /* stresses have been calculated at the node          */
      p = *(elnode + i);                  /* node number      */
      q = valu(sign, p) + 1;              /* number for average   */
      storeu(sign, p, q);
      stored(sigx, p, ((q-1) * vald(sigx, p) + sig[0])/q);
      stored(sigy, p, ((q-1) * vald(sigy, p) + sig[1])/q);
      stored(sigxy, p, ((q-1) * vald(sigxy, p) + sig[2])/q);
    }
                          /* end of loop for one node     */
  }                       /* end loop forone element      */
  close(input);
_oldfile = 99;
  scr_clr();
  return;
}

/*=====================================================================
          ******************************
          *   PRINT STRESS VECTORS     *
          *   prntstrs(dim)            *
          ******************************/
prntstrs(dim, displace)
  unsigned dim;
float *displace;
{
  int i, output;
```

```
   char a;
   char outfile[100];
   static char extout[5] = ".ot";             /* output file extension */
   extern unsigned matrix;
   unsigned sigx, sigy, sigxy, sign;

   sigx = matrix;                     /* paragraphs for stress vectors    */
   sigy = matrix + (dim+6) / 2;
   sigxy = matrix + 2 * ((dim+6) / 2);
   sign = matrix + 3 * ((dim+6) / 2);

   strcpy(outfile, file);
   strcat(outfile, extout);

   while (!(output = creat(outfile))){
       printf("\n Cannot open %s", outfile);
       printf("\n Press <E> to exit, any other to try again");
       if((a = ci()) == 'e' || a == 'E')
           exitl(0);
   }
   printf("Writing to output file: %s", outfile);

   for (i = 1; i <= dim; i++){
       if(valu(sign, i)){      /* if there were any stresses at this node */
           fprintf(output, "%5d%15.6E%15.6E%15.6E\n", i, vald(sigx, i),
                                           vald(sigy, i), vald(sigxy, i));
       }
   }
   close(output);
_oldfile = 99;


output = creat("disp");
for (i = 1; i <= dim; i++)
   fprintf(output, "%5d%15.4E%15.4E%15.4E\n", i, *(displace+2*i-1),
                       *(displace+2*i), 0.0);

close(output);
_oldfile = 99;
   return;
}

/*=============================================================
                   **************************
                   *   ENTER FILE NAME      *
                   *       getname()        *
                   **************************                  */
getname()
{
extern char file[];
extern char istress;
extern int nint, nintt;
extern double ntc[][2];

   scr_rowcol(10, 10);
   printf("Enter name of input files: ");
   scanf("%s", file);

   scr_rowcol(12, 10);
   printf("Enter <1> for plane stress or <0> for plane strain: ");
   while (((istress = ci()) != '0') && (istress != '1'))
       ;

   scr_rowcol(14, 10);
   printf("Enter number of integration points - (2 to 4)");
   while((nint = ci() - '0') < 2 || nint > 4)
       ;
   scr_clr();
   switch(nint){
       case 2:
           nintt = 0;
           break;
       case 3:
           nintt = 2;
           break;
       case 4:
           nintt = 5;
           break;
   }

   return;
}

/*====================================================================
                      ***************************************
                      *   INETGER COMPARISON FUNCTION       *
                      *        test = intcmp(i,j)           *
                      * if i > j  --->  test > 0            *
                      ***************************************          */

int intcmp(i, j)        /* compares ints i and j for qsort  */
int *i, *j;
{
   return(*i - *j);
}

/*=========================================================================
             ***************************************
             *   EXIT ROUTINE THAT FIXES SCREEN   *
             ***************************************/

exitl(n)
int n;
{
   ci();
   screen(0);
   width(80);
   scr_curson();
   exit(n);
```

```
/*

FILE : FINEL1.C */
/*                  ROUTINES FOR OBLIQUE BOUNDARIES
                    AND FRICTIONLESS INTERFACES

Procedure:
            a)  Read redirection information with formredir().
                Then any reference to a given node will be
                redirected to that which is specified.

            b)  Read boundary transformations with formtrans().
                When elements are transformed, first displacement
                is parallel to angle.

            c)  When forming each element, the element stiffness
                matrix is formed as usual, then rednode(elnode) is called
                to adjust elnode for redirection, then eltrans(elmatrix,
                elnode) is called to transform the element, then the
                element stiffness matrix is assembled into the global
                matrix as usual.

            d)  Once the global stiffness matrix and load vector
                are formed, the load matrix is transformed using
                loadtrans(load).

            e)  Now the system of equations are solved for the disp-
                placements, and then displacements common to more than
                one node are assigned properly using unredir(disp).

            f)  Finally the displacements are transformed back using
                disptrans(disp).  Now the stresses can be found as
                before.
*/

struct trans {
   unsigned node;
   float angle;
};
static struct trans *transform;        /* node transform array        */
static unsigned transfn;             /* number of nodes transformed  */
extern char file[];
extern unsigned diags;
unsigned *redir, redirn;
extern int _oldfile;

/*==================================================================
     *********************************************
     *  GET ANGLES FOR BOUNDARY TRANSFORMATIONS *
     *********************************************/

formtrans()
{
    unsigned i, j, node, n;
    unsigned input;
    float angle;
    static char infile[20];          /* file for input       */
    double pi();

    strcpy(infile, file);
    strcat(infile, ".tns");

    if ((input = open(infile, 0)) == 0){/* if there are no nodes to be trans*/
        transfn = 0;
        return;
    }
    printf("\nReading Transformation File %s", infile);

    fscanf(input, "%u", &transfn);      /* read number to be transformed*/
    if (!transfn)
        return;

    if(!(transform = calloc(transfn, sizeof(struct trans)))){
        printf("\nNo room for tranformation array");
        exitl();
    }

    n = 0;
    for(i = 0; i < transfn; i++){
        if(!fscanf(input, "%u%f", &node, &angle))  /* get node and angle   */
            break;
        for (j = 0; j < redirn; j++){               /* check for redirection*/
            if (node == (*(redir+2*j)+1)/2){        /* if there is redirect */
                node = (*(redir + 2*j+1)+1)/2;      /* transform node redir */
                break;
            }
        }
        for (j = 0; j <= n; j++)                  /* make sure no duplicates  */
            if (node == (transform+j)->node){
                node = 0;
                break;
            }

        if (node){                                  /* if there was no redir*/
            (transform+i)->node = node;                 /* assign node and  */
            (transform+i)->angle = pi() * angle / 180.0;/* angle in radians */
            n++;
        }
    }
    transform = realloc(transform, n*sizeof(struct trans));
    transfn = n;
    close(input);
  _oldfile = 99;
  }

  /*══════════════════════════════════════════════════════════════
                ******************************
                *   GET REDIRECTION VECTOR   *
```

```
               *****************************/
formredir()
{
   unsigned i, j, input, n, n1, n2;
   static char infile[20];
   extern unsigned diags;
   static char status[10];           /* status character to see what */
                                     /* gets linked together.        */
                                     /* b - both   f - first   s - second   */

   strcpy(infile, file);
   strcat(infile, ".red");

   input = open(infile, 0);
   if (input == 0){
       redirn = 0;
       return;
   }
   printf("\nReading Redirection File %s", infile);

   fscanf(input, "%u", &redirn);   /* get number of linked nodes    */
   if (!redirn)
       return;

   if(!(redir = calloc(redirn*4, sizeof(unsigned)))){
       printf("\nNo room for redirection array");
       exit1();
   }

   n = 0;
   for (i = 0; i < redirn; i++){
       if(!fscanf(input, "%u%u%s", &n1, &n2, status))
           break;
       switch (toupper(*status)){
           case 'B':
               *(redir+n) = 2*n1-1;
               *(redir+n+1) = 2*n2-1;
               *(redir+n+2) = 2*n1;
               *(redir+n+3) = 2*n2;
               storeu(diags, 2*n1-1, 0);
               storeu(diags, 2*n1, 0);
               n += 4;
               break;
           case 'F':
               *(redir+n) = 2*n1-1;
               *(redir+n+1) = 2*n2-1;
               storeu(diags, 2*n1-1, 0);
               n += 2;
               break;
           case 'S':
               *(redir+n) = 2*n1;
               *(redir+n+1) = 2*n2;
               storeu(diags, 2*n1, 0);
               n += 2;
               break;
           default:
            printf("\nError in redirection data.");
            exitl(0);
        }
    }
    redirn = n/2;
    if(!(redir = realloc(redir, 4*redirn))){
        printf("\nError in reallocation in formredir");
        exitl();
    }
    close(input);
_oldfile = 99;
    return;
}

/*=====================================================================
              ***************************************
              *   ADJUST ELNODE FOR REDIRECTION     *
              ***************************************/
rednode(elnode)            /* use elnode adjusted to 16 unsigned elements  */
unsigned *elnode;      /* in it                                            */
{
    int i, j;

    if (!redirn)       /* if there is no redirection        */
        return;

    for (i = 0; i < 16; i++)              /* for each elnode        */
        for(j = 0; j < redirn; j++)       /* check each row of redir  */
            if (*(elnode + i) == *(redir + 2*j)){/* if this is redirected   */
                *(elnode + i) = *(redir+2*j+1); /* set redirected node     */
                break;
            }
    return;
}

/*=====================================================================
              *******************************************
              *   TRANSFORM ELEMENT STIFFNESS MATRIX    *
              *   FOR NODE DIRECTION                    *
              *******************************************/
eltrans(elmatrix, elnode)
unsigned *elnode;          /* elnode has been adjusted to 16 and redirected*/
double *elmatrix;          /* elmatrix is stored in lower triangular form  */
{
    unsigned i, j;
    static float angles[8];
    static double ut[8];       /* upper triangular elements lost in trans  */
    double temp1, temp2, angle;
    unsigned m1, m2;           /* pointers to start of row pairs           */
    double sin(), cos();

    if (!transfn)              /* if no nodes transformed        */
        return;

    for (i = 0; i < 8; i++)
```

```
        *(angles+i) = 0.0;                  /* clear angles */

    for (i = 0; i < 16; i++){               /* for each node in elnode      */
        for (j = 0; j < transfn; j++){      /* check all nodes for transf   */
            if ((*(elnode+i)+1)/2 == (transform+j)->node){
                *(angles + i/2) = (transform+j)->angle;
                break;
            }
        }
    }
                                            /* angles has been formed       */
                                            /* now do [T][K]                */
    m1 = 0, m2 = 1;                         /* start of first an second rows*/
    for (i = 0; i < 8; i++){                /* for each pair of rows        */
        *(ut+i) = *(elmatrix+m2+2*i);       /* set lost upper element       */
        if (angle = *(angles + i)){         /* if this row is transformed   */
            for (j = 0; j <= 2*i; j++){     /* then for each column in rows */
                temp1 = *(elmatrix+m1+j);   /* must keep values in columns  */
                temp2 = *(elmatrix+m2+j);   /* then assign new values       */
                *(elmatrix+m1+j) = cos(angle) * temp1 + sin(angle) * temp2;
                *(elmatrix+m2+j) = cos(angle) * temp2 - sin(angle) * temp1;
            }
            temp1 = temp2;                  /* ready for diagonal element in*/
            temp2 = *(elmatrix+m2+j);       /* second row and lost upper    */
            *(ut+i) = cos(angle) * temp1 + sin(angle) * temp2;
            *(elmatrix+m2+j) = cos(angle) * temp2 - sin(angle) * temp1;
        }
        m1 += 4*i + 3;                      /* set indeces to next row pair */
        m2 += 4*i + 5;
    }
                                            /* [T][K] is complete - now do
                                               ([T][K])[T]transp            */
    m1 = 0;                                 /* set top of column            */
    for (i = 0; i < 8; i++){                /* for each column pair         */
        if (angle = *(angles + i)){         /* if this was a transformed col*/
            *(elmatrix+m1) = *(elmatrix+m1)*cos(angle) + *(ut+i) * sin(angle);
            m2 = 0;                         /* initialize offset from diag  */
            for (j = 2*i+1; j < 16; j++){   /* for rest of rows in column   */
                m2 += j;
                temp1 = *(elmatrix+m1+m2);
                temp2 = *(elmatrix+m1+m2+1);
                *(elmatrix+m1+m2) = temp1*cos(angle) + temp2*sin(angle);
                *(elmatrix+m1+m2+1) = temp2*cos(angle) - temp1*sin(angle);
            }
        }
        m1 += 4*i+5;
    }
    return;
}

/*══════════════════════════════════════════════════════════════════════
            ***************************************
            *   TRANSFORM LOAD VECTOR - [T][F]    *
            ***************************************/
loadtrans(load)
```

```
float *load;
{
   unsigned i;
   float *p, angle, temp;

   for (i = 0; i < redirn; i++)          /* first redirect loads      */
       *(load + *(redir+2*i+1)) += *(load + *(redir+2*i));

   if (!transfn)                         /* if no transformations     */
       return;                           /* return                    */

   for (i = 0; i < transfn; i++){        /* transform loads at nodes */
       p = (load + 2 * ((transform+i)->node)-1);
       angle = (transform+i)->angle;
       temp = *p;
       *p = *p * cos(angle) + *(p+1) * sin(angle);
       *(p+1) = -temp * sin(angle) + *(p+1) * cos(angle);
   }
   return;
}

/*
          ****************************************************
          *   TRANSFORM DISPLACEMENT VECTOR BACK  [T]inv[X]   *
          ****************************************************/
disptrans(disp)
float *disp;
{
   unsigned i, j, offset;
   float *p, angle, temp;

   if (!transfn)                         /* if no transformations     */
       return;

   for (i = 0; i < transfn; i++){
       p = (disp + 2 * (transform+i)->node - 1);
       angle = (transform+i)->angle;
       temp = *p;
       *p = *p * cos(angle) - *(p+1) * sin(angle);
       *(p+1) = temp * sin(angle) + *(p+1) * cos(angle);
       for (j = 0; j < redirn; j++){           /* transform redirected pairs*
           /* if first in redirected pair matches transformed load        */
           if (*(redir+2*j+1) == 2 * (transform+i)->node - 1){
               p = (disp + *(redir+2*j));      /* find which was redirect  */
               temp = *p;
               *p = *p * cos(angle) - *(p+1) * sin(angle);
               *(p+1) = temp * sin(angle) + *(p+1) * cos(angle);
               if (*(redir+2*j) + 1 == *(redir+2*j+2))/* don't transform  */
                   j++;                                /* twice           */
           /* else if second in redirected pair matches transformed load  */
           } else if (*(redir+2*j+1) == 2 * (transform+i)->node){
               p = (disp + *(redir+2*j) - 1);  /* find which was redirect  */
               temp = *p;
               *p = *p * cos(angle) - *(p+1) * sin(angle);
               *(p+1) = temp * sin(angle) + *(p+1) * cos(angle);
```

```
            }
        }
    }
    return;
}

/*======================================================================
               *************************************
               *   UNREDIRECT DISPLACEMENT VECTOR  *
               *************************************/
unredir(disp)
float *disp;
{
    unsigned i, *m;

    if (!redirn)                           /* if no redirections   */
        return;

    m = redir;
    for (i = 0; i < redirn; i++, m+=2)
        *(disp + *m) = *(disp + *(m+1));
}

/*===========================================================================
               FIND THE n'TH OCCURENCE REDIRECTED TO NODE                  */
unsigned backred(node, n)
{
unsigned i = 0;                /* index for rednode
unsigned val = 0;          /* valu to be returned                          */

    if (!n)                /* if looking for zeroth occurence              */
        return(0);         /* not defined                                  */
    for (i = 0; i < redirn && n; i++)  /* go thru redirected nodes once    */
        if (*(redir + 2*i + 1) == node){/* if this node redirected to node */
            n--;                       /* decrement n                      */
            val = *(redir + 2*i);      /* set the value                    */
        }

    if (!n)                            /* if found the nth occurence       */
        return(val);                   /* return the proper node           */

    return(0);                         /* if not found return zero         */
}
```

```
/*

FILE : FMATRIX.C   */

#include "mylib.h"
#include "stdio.h"
unsigned findij();
unsigned matrix, diags;          /* stiff matrix, diagonal numbers   */
unsigned valfu(), backred();
extern int _oldfile;
/*==========================================================================
                      INITIALIZE THE MATRIX                              */
initialize(dim)                      /* initializes matrix */
   unsigned dim;                  /* dimension of matrix */
{
   unsigned i;          /* counter and partition number */
   unsigned sysmem(), _showds(), avail;

   avail = 161 * (sysmem() * 641 - _showds() - 0x1000)/61;
   avail -= dim/3 + 5;                   /* availbl matrix space     */
   diags = extalloc((long)dim * 21);        /* get diagonals array          */
   matrix = extalloc((long)avail * 61);          /* stiffness matrix
   if(!matrix || !diags){
       printf("\nNo room for matrix.");
       exitl(0);
   }
   storeu(diags, 0, avail);        /* store available structures in diags  */
   for (i = 1; i <= dim;  i++)     /* store non-partitioned flag            */
       storeu(diags, i, 1);

   return;                         /* we're all ready to go                 */
}

/*===========================================================================
                      PREPARE STIFFNESS MATRIX                             */
formstiff(dim)
unsigned dim;                                /* dimension of system        */
{
   int input;
   unsigned i, j, k, l, m, n;                /* counters                   */
   unsigned nel, nell;                       /* elements, elements in file */
   unsigned node, upper;
   unsigned diag;                            /* structure for diagonal value */
   unsigned fnode, nred;                     /* node to search, redirected # */
   unsigned *elnode, *nodes;                 /* element node numbers       */
   static unsigned tnodes[16];
   unsigned top = 1;                         /* top of matrix stack        */
   static char nodefile[100];                /* name of node files         */
   static char filenum[3] = "0";             /* overlay numbers start at 1 */
   extern char file[];                       /* model file names           */
   static double row[100];                   /* temporary storage for a row */

   elnode = malloc(0);                       /* initialize elnode          */
   nel = 0;                                  /* initialize number of element */
```

```
i = j = 0;                                 /* set counters                    */
while (1){                                 /* start loop to read nodes        */
    (*filenum)++;                          /* get next overlay                */
    strcpy(nodefile, file);                /* copy filename to nodefile       */
    strcat(nodefile, ".el");               /* add extension                   */
    strcat(nodefile, filenum);             /* add overlay number              */
    if ((input = open(nodefile, 0)) < 1)      /* if no such file              */
        break;                                /* exit loop                    */

    printf("\nReading overlay %s.", nodefile);
    fscanf(input, "%u", &nell);            /* get number of elements          */
    printf("\n  Elements in file = %u", nell);
    i = nel;                               /* start of elnode for file        */
    nel += nell;                           /* new number of elements          */
    elnode = realloc(elnode, 16 * nel);    /* realloc elnode for new file     */
    if (!elnode){
        printf("\nNo room for elements");
        exitl(0);
    }
    for (; i < nel; i++){          /* for each element in [illegible]          */
        nodes = elnode + 8 * i;            /* get pointer to next element     */
        fscanf(input, "%u%u%u%u%u%u%u%u%u%*F%*F", &node, nodes, nodes+1, n
                       nodes+3, nodes+4, nodes+5, nodes+6, nodes+7);
        for (k = 0; k < 8; k++)            /* check for errors                */
            if((*(nodes+k) > dim) || (*(nodes+k) < 1)){
                printf("\nError in %s.  Element %u", nodefile, node);
                printf("\n%u %u  %u %u  %u %u", k-1, *(nodes+k-1),
                        k, *(nodes+k), k+1, *(nodes+k+1));
                exitl(0);
            }
    }
    close(input);                          /* close this overlay file         */
    _oldfile = 99;
}
                                           /* nodes are all read              */
printf("\n\nForming matrix.");
printf("\nMatrix structures available - %u", valu(diags, 0));
upper = nel * 8;                           /* last node number in elnode      */
for (i = 1; i <= 2*dim; i++){              /* for each row of matrix          */
    scr_rowcol(24, 5);
    printf("%-5u", i);
    if (!valu(diags, i))                   /* if partitioned row              */
        continue;                          /* continue with next row          */

    diag = top++;
    storeu(diags, i, diag);        /* else store top in diag          */
    storefu(diag, top - diag);     /* store number of elements in row */
    fnode = (i+1)/2;               /* fnode is the node to look for   */
    nred = 1;                      /* number of nodes redirected to i */
    do(                            /* start looking for node          */
        for (j = 0; j < upper; j++){   /* look for row in each node       */
            if (*(elnode+j) == fnode){ /* if node is found in element     */
                for (nodes = elnode + j-j%8, k = 0; k < 8; k++){
                    tnodes[2*k] = 2 * *(nodes+k) - 1;   /* get element    */
                    tnodes[2*k+1] = 2 * *(nodes+k);     /* global #'s     */
```

```
                   }
                   rednode(tnodes);                          /* do reassignments */
                   for (k = 0; k < 16; k++){
                       if ((l = tnodes[k]) > i && !findij(i, l)
                       && valu(diags, l)){ /* if in upper tri and not part */
                           storefu(top++, l);  /* store j index            */
                           storefu(diag, top - diag);/* store number in row*/
                       }
                       if (top == valu(diags, 0)){     /* if out of matrix sp
                           printf("\nOut of matrix space");
                           exitl(0);
                       }
                   }
               }
           }                                /* loop until no more   */
       } while (fnode = (backred(i, nred++)+1)/2); /* redirections to i    */
/*diag = valu(diags, i);
printf("\n\nrow %u", i);
printf("  num %u", valfu(diag));
for (j = 1; j < valfu(diag); j++)
printf("%5u", valfu(diag+j));*/
   }
   printf("\nMatrix structures used - %u.", top);
   ci();
   if(!free(elnode)){                          /* release storage for later    */
       puts("cannot free elnode in formstiff");
       exitl(0);
   }
   for  (i = 0; i < top; i++)                  /* zero out stiffness matrix    */
       storefd(i, 0.0);

}

/*==============================================================================
                          FIND STRUCTURE FOR I, J                              */
unsigned findij(i, j)
unsigned i, j;
{
   unsigned k, l;
   extern unsigned matrix, diags;

   if (!valu(diags, i) || !valu(diags, j)) /* if partitioned row, column    */
       return(0);

   if (i < j){                /* if caller wants it from upper triangle    */
       k = valu(diags, i);    /* get diagonal element */
       for (l = 1; l < valfu(k); l++)  /* check each one after for match   */
           if (valfu(k+l) == j)        /* if match was found               */
               return(k+l);            /* return the structure number      */
   }

   if (i == j)
       return(valu(diags, i));         /* return value from diagonal array */

   if (i > j){          /* if caller wants element in lower tri     */
```

```
       k = valu(diags, j);     /* get diagonal element                   */
       for (l = 1; l < valfu(k); l++)
           if (valfu(k+l) == i)
               return(k+l);
   }
   return(0);                                  /* print mess, return zero  */
}

/*==================================================================
           **********************************************
           *   ASSEMBLE AN ELEMENT STIFFNESS MATRIX     *
           *   INTO THE GLOBAL STIFFNESS MATRIX         *
           *   ELEMENT STIFFNESS STORED SEQUENTIALLY    *
           *   IN LOWER TRIANGULAR FORM.                *
           *   assemble(elmatrix, node_numbers, eldim)  *
           **********************************************/

assemble(elmatrix, glbnode, dim)   /* assembles element */
                                   /* stiffness matrix into global */
                                   /* elmatrix in lower triangle   */
   unsigned dim;           /* dimension of element matrix */
   double *elmatrix;       /* contains element stiffness matrix*/
   unsigned *glbnode;      /* contains nodes for element */
{
   unsigned i, j, k, l, n;
   unsigned findij();

   for (i = 0; i < dim; i++){      /* for each row of element */
       for (j = 0; j <= i; j++){   /* for each column in element */
           k = *(glbnode+i);       /* first ndex for global */
           l = *(glbnode+j);       /* second ndex for global   */
           if(valu(diags, k) && valu(diags, l)){   /* if not partitioned   */
               if (n = findij(k, l))           /* get matrix number        */
                   addfd(n, *elmatrix);        /* add value from element   */
               else                            /* print mess if not found  */
                   printf("\nCannot find A[%u, %u]", k, l);
           }
           elmatrix++;
       }
   }
   return;
}
```

```
/*

FILE : LOAD.C      */
/*
           ********************************************
           *       Finite Element Routine             *
           *   Two Dimensional Stress Analysis        *
           *     Plane Stress - Plane Strain          *
           *    Load vector generation program        *
************                                        *********
*  Input is in three parts:                                 *
*      1) nodecrd  -   has coordinates for each node its    *
*                      first line has the number of nodes   *
*      2) file.ldN  -  each line contains three integers    *
*                      corresponding to the three global    *
*                      nodes for each of the element nodes. *
*      4) file.trN  -  This contains the stresses.          *
*                      Each line contains a node number,    *
*                      a force in the x direction and a     *
*                      force in the y direction.            *
*************************************************************/

#include "c:math.h"
#include "c:stdio.h"
extern int _oldfile;

   extern double N();                  /* N() = interpolation function */
   extern char file[20];               /* file names for I/O           */
   extern char filenum[2];             /* overlay number extension */
   extern float *nodecord;
/*==============================================
           ********************************************
           *   FORM LOAD VECTOR AND ADD TO FORCE      *
           *          load(force, thick,, dim)        *
           ********************************************/

load(force, thick, dim)                /* generate load vector */
   float *force, *thick;
   int dim;
{
   int input, *node, i;
   extern char file[];
   extern char filenum[];
   static char loadfile[100];
   int node[3];
   double strs[6];
   double elforce[6];

   strcpy(loadfile, file);
   strcat(loadfile, ".ld");
   strcat(loadfile, filenum);      /* add overlay extension    */


   if ((input = open(loadfile, 0)) <= 0){
       printf("\nCannot find tractions data file - %s.", loadfile);
       exitl(0);
   }

   printf("\nReading boundary tractions file: %s ", loadfile);

   while ((i = fscanf(input, "%d%F%F%d%F%F%d%F%F", node, strs, strs+1,
                                     node+1, strs+2, strs+3,
                                     node+2, strs+4, strs+5)) > 5){
       elload(node, strs, thick, elforce);
       adload(force, elforce, node);
   }

   close(input);
_oldfile = 99;
   return;
}

/*=================================================================
          **************************************************
          *   ELEMENT LOAD VECTOR GENERATOR                 *
          *  elload(node, strs, thick, elforce)             *
          **************************************************/

elload(node, strs, thick, elforce)     /* generate element force vector */
   double *elforce, *strs;
   float *thick;
   int *node;
{

   int i, j;
   double N();          /* use two dimensional interp. fns in terms
                                of y with x = 1          */
   double x, weight, dx, dy, thickness;

   for (i = 0; i < 6; i++)
       *(elforce + i) = 0.0;            /* empty element force vector */

   for (i = 0; i < 2; i++){         /* at each integration point */
       x = (i) ? -0.577350269189626 : 0.577350269189626;
       weight = 1.0;


       dx = dy = thickness = 0.0;
       for (j = 0; j < 3; j++){     /* get dx, dy/deta  */
           dx += N(j, 2, 1.0, x) * *(nodecord + 2 * *(node + j));
           dy += N(j, 2, 1.0, x) * *(nodecord + 2 * *(node + j) + 1);
           thickness += N(j, 0, 1.0, x) * *(thick + *(node + j));
       }

       dx = sqrt(dx * dx + dy * dy) * weight * thickness;
/*     ds  =   sq(dx^2 + dy^2)      *  weight   * thickness          */

       for (j = 0; j < 3; j++){     /* do {N}[T]dx and add to force */
           *(elforce + 2*j) += dx * N(j, 0, 1.0, x) * *(strs + 2 * j);
```

```
            *(elforce + 2*j + 1) += dx * N(j, 0, 1.0, x) * *(strs + 2 * j + 1)
        }
    }
    return;
}
/*=============================================================
      ***********************************************
      *   ROUTINE TO ADD ELEMENT LOAD TO GLOBAL LOAD  *
      *       adload(force, elforce, node)            *
      ***********************************************/

adload(force, elforce, node)       /* add element force to load */
   double *elforce;
   float *force;
   int *node;
{
   int i;

   for (i = 0; i < 3; i++){
       *(force + 2 * *(node + i) - 1) += *(elforce + 2 * i);
       *(force + 2 * *(node + i)) += *(elforce + 2 * i + 1);
   }
   return;
}

/*==============================================================
          ****************************************
          *   ROUTINE TO ADD NODAL POINT LOADS    *
          *  pload(loadv)                         *
          ****************************************/
pload(loadv)
float *loadv;
{
extern char file[];                  /* data file name              */
static char pldfile[20];
static char ext[5] = ".pld";     /* data file extension         */
int input;                       /* file specifier              */
unsigned node;                   /* node number read from file  */
double xload, yload;

   strcpy(pldfile, file);        /* form name of input file     */
   strcat(pldfile, ext);
   scr_clr();
   scr_rowcol(10, 20);


   if((input = open(pldfile, 0)) != NULL && input != ERR){
       while(fscanf(input, "%u%F%F", &node, &xload, &yload) != EOF){
           *(loadv + 2*node - 1) += xload;
           *(loadv + 2*node) += yload;
       }
   }
   close(input);
_oldfile = 99;
   return;
```

```
/*


FILE : GSSOLVE.C        */
/*              ******************************************
                *   GAUSS - SIEDEL iteratIVE SOLVER *
                *    FOR LINEAR SYSTEM OF EQUATIONS      *
                *      STORED IN SPARSE MATRIX FORM      *
                ******************************************          */

#include "math.h"
#include "mylib.h"
double weight;
int converge();
extern unsigned matrix, diags;
unsigned valfu();
double valfd();
/*===================================================================
                   ************************************
                   *            SOLVER ROUTINE          *
                   *  solve(disp, dim)                  *
                   *  disp - pointer to double pre-     *
                   *  cision forcing vector.  It also   *
                   *  contains the response vector on   *
                   *  exit.                             *
                   *  dim - the dimension of the sys-   *
                   *  tem of equations, option base 0   *
                   ************************************          */
solve(disp, dim)        /* solver using Gauss - Seidel iteration */
   float *disp;     /* disp holds the force vector on entrance */
   unsigned dim;    /* dim is the dimension of the system
                       option base zero */
(
/*-------------------------------------------------------------------
DECLARE VARIABLES                                                */

   float *f;            /* force vector will be held here        */
   float *xn, *xnpl;    /* xn and xnpl hold nth and n+lth iterations   */
   double tolerance;    /* tolerance for change in diplacement vector  */
   int check;           /* flag to indicate if displacement
                           within tolerance */
   long itt;            /* iteration count  */
   extern double weight;   /* weight for over relaxation */
   unsigned temp;
   int iterate();
   int i, conmax(), n;
   char control, printflag, conflag;
   double timer(), tl, t2;      /* used for timing the solver    */
   static int numb[6];
   double temp;
/*-------------------------------------------------------------------
ALLOCATE WORKING VECTORS                                         */

   if (!(f = calloc(dim + 1, sizeof(float))) ||
       !(xnpl = calloc(dim+1, sizeof(float))))(
        printf("\nCannot form workspace in solver");
        exitl(0);
    }

    xn = disp;                          /* set start of displacemant vector */
    syscheck(dim);                      /* check stability of system    */

    numb[0] = 1;                        /* get row numbers for printx */
    while(!valu(diags, numb[0]))
        numb[0]++;
    numb[4] = dim;
    while(!valu(diags, numb[4]))
        numb[4]--;
    numb[2] = (numb[4] + numb[0])/2;
    numb[3] = (numb[4] + numb[2])/2;
    numb[1] = (numb[0] + numb[2])/2;
    for(i = 1; i < 4; i++)
        while(!valu(diags, numb[i]))
            numb[i]++;

    for (i = 1; i <= dim; i++){
        *(f + i) = *(xn + i);           /* copy forcing vector to f */
    }

    _setmem(xn, (dim+1)*4, 0);  /* empty displacement vectors */
    _setmem(xnpl, (dim+1)*4, 0);    /* empty displacement vectors */

/*-------------------------------------------------------------------
START iteratIONS                                                  */

    check = 1;                      /* set nonconverge flag on          */
    conflag = 0;                    /* set converge check flag on       */
    printflag = 0;                  /* set display flag on              */
    control = 0;                    /* set control to data entry        */
    tolerance = 0.000001;           /* set default tolerance            */
    weight = 1.8;                   /* set default weight               */
    scr_cursoff();

    itt = 01;                       /* number of iterations */
    solv_scr(numb);                 /* generate solver screen    */
    scr_scrup(0, 21, 0, 24, 80);            /* clear bottom window      */
    scr_rowcol(22, 10);                     /* print instructions       */
    printf("<E>xit Solver                  <N>ew Tolerance or Weight");
    scr_rowcol(23, 10);
    printf("<C>onvergence Toggle           <D>isplay Toggle");
    scr_rowcol(5, 18);
    printf("%-5.2F", weight);               /* printf weight            */
    scr_rowcol(5, 36);
    printf("%8.6F", tolerance);             /* printf tolerance         */
    tl = timer();                   /* start time               */

    while (check){      /* while not converged  */
        switch (control) {
        case 0:                                 /* if no key has been pressed   */
            temp = xnpl;                        /* swap xnpl and xn */
        xnpl = xn;
        xn = temp;
        scr_rowcol(5, 62); ptimer(timer() - t1);/* print iteration time*/
        scr_rowcol(7, 65); printf("%D", itt++); /* print iterations */

        iterate(xn, xnpl, f, dim);                  /* assembler routine     */
        if (conflag){                               /* if converge flag set */
            numb[5] = conmax(xn, xnpl, dim);        /* check convergence     */
            check = (fabs((xn[numb[5]] - xnpl[numb[5]])/xnpl[numb[5]])
                     <= tolerance) ? 0 : 1;         /* see if converged      */
        }
        if (printflag)                              /* if printflag set      */
            printx(xn, xnpl, dim, numb);        /* display converg progress */
        break;                                      /* end case 0            */

    case 'E':                                   /* if control says end  */
        check = 0;                              /* set end flag         */
        break;                                  /* end case 'E'         */

    case 'Q':
        exit1(0);
        break;

    case 'D':                                   /* if conv display toggle   */
        scr_scrup(0, 9, 17, 19, 50);            /* clear display area       */
        printflag = 1 - printflag;              /* toggle printflag         */
        break;                                  /* end display toggle       */

    case 'C':                                   /* if converg check toggle  */
        scr_scrup(0, 19, 10, 19, 50);
        conflag = 1 - conflag;                  /* toggle conflag           */
        if (!conflag){                          /* if flag is off           */
            scr_rowcol(19, 20);
            printf("Convergence Test Off");
            numb[5] = 0;                        /* set max node to 0        */
        }
        break;

    case 'N':                                   /* new tolerance and weight */
        t2 = timer();                           /* account for input time   */
        scr_scrup(0, 21, 0, 24, 80);            /* clear bottom window      */
        scr_rowcol(22, 15);
        printf("Enter iteration residual weight( 0 to 2 )(%4.2F): ", weigh
        scanf("%F", &temp);                     /* enter new weight         */
        weight = (temp) ? temp : weight;        /* make assignment          */
        scr_rowcol(5, 18);
        printf("%-5.2F", weight);

        scr_scrup(0, 21, 0, 24, 80);            /* clear bottom window      */
        scr_rowcol(22, 15);
        printf("Enter tolerance(%7.5F): ", tolerance);
        scanf("%F", &temp);                     /* enter new tolerance      */
        tolerance = (temp) ? temp : tolerance;
        scr_rowcol(5, 36);
        printf("%8.6F", tolerance);
```

```
            scr_scrup(0, 21, 0, 24, 80);            /* clear bottom window    */
            scr_rowcol(22, 10);
            printf("<E>xit Solver                  <N>ew Tolerance or Weight");
            scr_rowcol(23, 10);
            printf("<C>onvergence Toggle           <D>isplay Toggle");
            tl += (timer() - t2);                   /* account for input time  */
            break;
        }                                           /* end switch              */

        if (check){                                 /* if not converged        */
            control = toupper(csts());              /* get new control         */
        } else {                                    /* else if converged       */
            t2 = timer();                           /* account for entry time  */
            scr_scrup(0, 21, 0, 24, 80);            /* clear bottom window     */
            scr_rowcol(21, 20);
            printf("iterations = %D    time to solve: ", itt);
            ptimer(t2 - tl);                        /* print time to solve     */
            scr_rowcol(23, 20);
            printf("Press <A> to change tolerance, any other key to return.");
            while(!(control = csts()) && (timer() - t2) <= 60.0)
                ;                           /* wait for input for 10 minutes   */
            switch(toupper(control)){
            case 'A':                       /* if more iterations wanted       */
                check = 1;                  /* set nonconverge flag            */
                control = 'N';              /* set flag for new tolerance      */
                tl += timer() - t2;         /* adjust for time for input       */
                break;
            default:
                break;
            }
            csts();                         /* clear keyboard buffer           */

            tl += (timer() - t2);                   /* account for input time  */
        }                               /* end if - else                   */
    }                                   /* end while loop                  */

    free(f);
    if (xnpl != disp){                  /* if displacement is not in disp  */
        for (i = 1; i <= dim; i++)
            *(disp + i) = *(xnpl + i);
        free(xnpl);
    } else {
        free(xn);
    }
    scr_curson();
    return;
}
/*
                    ******************************
                    *   PRINT GRAPHICS SCREEN    *
                    ******************************/
solv_scr(numb)
unsigned numb[];
{
   unsigned i;

   scr_clr();
   scr_rowcol(2, 25); printf("* SPARSE MATRIX SOLVER *");
   scr_rowcol(1, 25); printf("**************************");
   scr_rowcol(3, 25); printf("**************************");
   scr_rowcol(4, 7);
   printf("|---------------------------------------------------------------|"
   scr_rowcol(5, 7);
   printf("| weight =       tolerance =           iteration time:          |"
   scr_rowcol(6, 7);
   printf("|--------|---------------|------------------|-------------------|"
   scr_rowcol(7, 7);
   printf("|   i    |      x(i)     |     change       | iteration:        |"
   scr_rowcol(8, 7);
   printf("|--------|---------------|------------------|-------------------|"
   for (i = 9; i < 20; i++){
       scr_rowcol(i, 7); putchar('|');
       scr_rowcol(i, 71); putchar('|');
   }
   for(i = 0; i < 5; i ++){             /* print row numbers for printx */
       scr_rowcol(9 + 2*i, 10);
       printf("%-5d", numb[i]);
   }
   scr_rowcol(20, 7);
   printf("|--------|---------------|------------------|-------------------|"
   scr_rowcol(19,60);
   printf("Max Change.");

}

/*=======================================================================
                   **********************************
                   *   PRINT CONVERGENCE PROGRESS   *
                   *  printx(xi, xi+1, dim)         *
                   **********************************               */

printx(xn, xnpl, dim, numb)
   float *xn, *xnpl;
   unsigned dim, numb[];
{
   int i;

   for(i = 0; i < 5; i ++){
       scr_rowcol(9 + 2*i, 16);
       printf("%14.5e", xnpl[numb[i]]);
       if(xnpl[numb[i]]){
           scr_rowcol(9 + 2*i, 34);
           printf("%14.7E", ((1.0 - (xn[numb[i]])/xnpl[numb[i]])));
       }
   }
   if (numb[5]){
       scr_rowcol(19, 10);
       printf("%-5d", numb[5]);
       scr_rowcol(19, 16);
```

```
        printf("%14.5e", xnpl[numb[5]]);
        if(xnpl[numb[5]]){
            scr_rowcol(19, 34);
            printf("%14.7E", ((1.0 - (xn[numb[i]])/xnpl[numb[i]])));
        }
    }
}
/*===========================================================================
                    *********************************
                    *   CHECK STABILITY OF SYSTEM    *
                    *********************************/

syscheck(dim)
unsigned dim;
{
   unsigned i, n, j, flag, jj;

   scr_clr();
   scr_rowcol(2, 20);
   printf("Check System Stability\n\n");
   flag = 0;

   for (i = 1; i <= dim; i++){
       if(!(n = valu(diags, i)))        /* if partitioned node      */
           continue;                    /* continue with next row   */

       if (valfd(n) < 0.0){             /* if bad diagonal element in stiff */
           printf("\nNode %u diagonal is negative %F", i, valfd(n));
           getchar();                   /* print error message             */
           flag = 1;
       }
           if (valfd(n) == 0.0){
           printf("\nNode %u diagonal is zero and will be partitioned.", i);
           storeu(diags, i, 0);
       }
       for (j = 1; j < valfu(n); j++){ /* check rest of row       */
           jj = valfu(n+j);                         /* j index of element  */
           if(valfd(n) <= valfd(n+j)           /* if value is greater than */
           || valfd(valu(diags, jj)) <= valfd(n+j)){       /* diagonals    */
               flag = 1;                       /* set flag and             */
               printf("%11u%5u", i, jj);       /* print column of error*/
           }
       }
   }
  if (flag)
      getchar();
}
```

```
; File: iterat.a

; ******************************************************************
; *   Routine for solving a linear system of equations      *
; *   This portion is the portion that is itterated         *
; *   stored in sparse matrixform.  This routine utilizes *
; *   the 8087 math coprocessor.             *
; *   This routine is called from C as gssolve(x, dim)      *
; *   which calls itterate(xn, xnpl, forc, dim).  xn and xnpl *
; *   are single prec. displacement vectors, nth and n+1  *
; *   itterations.  xnpl is emptied on entry.         *
; *   forc is a pointer to a single precision forcing     *
; *   vector and dim is the dimension of the system.      *
; *   Reference "Numerical Methods" by Germund Dahlquist  *
; *   & Ake Bjorck.  trans. Ned Anderson. Pub. Prentice Hall  *
; *   page 188 sec. 5.6.                *
; *   Library Call no. QA 297 D13 E5              *
; ******************************************************************

DSEG
PUBLIC weight_          ;weighting factor for overrelaxation
PUBLIC matrix_, diags_

   xn  EQU     WORD [BP+4]
   xnpl    EQU WORD [BP+6]
   forc    EQU WORD [BP+8]
   dim EQU WORD [BP+10]
   row EQU WORD [BP-2]

CSEG
EVEN
PUBLIC valu_
PUBLIC iterate_
iterate_:
   PUSH BP
   MOV BP, SP
   SUB SP, 2

;-------------------------------------------------------------------
;EMPTY xnpl
   MOV CX, dim       ;get dimension of system
   INC CX            ;account for zero element
   SHL CX, 1         ;multiply CX by 2 to get number of words
   MOV DI, xnpl          ;get base of xnpl
   MOV AX, DS
   MOV ES, AX        ;get segment for xnpl
   MOV AX, 0         ;value to be stored in array
   REP STOSW         ;store 0 in xnpl

;-------------------------------------------------------------------
;START PASS 1 OF ITTERATION PROCESS
```

```
   MOV row, 0       ;set i
                  ;SI will be used for base of xn, xnpl, forc
                  ;DI will be used for stiffness matrix
                  ;BX will be used for offset of xn, xnpl, forc
startit:            ;start of loop one for one itteration
   INC row          ;increment row
   MOV AX, row      ;get row
   CMP AX, dim      ;see if row > dim
   JG endit         ;if greater, end this pass
   PUSH AX          ;check if partitioned
   PUSH WORD diags_    ;push diagonals paragraph
   CALL valu_       ;puts diagonal structure into ax
   ADD SP, 4        ;remove arguments from stack
   CMP AX, 0        ;see if partitioned
   JE startit       ;if partitioned loop back
   MOV CX, 6        ;6 bytes per element (single precision)
   MUL CX           ;get segment and offset for the row
   MOV CX, 16       ;16 bytes per paragraph
   DIV CX           ;AX has segment, DX has offset
   ADD AX, WORD matrix_    ;add base paragraph of matrix_
   ADD DX, 2        ;add information for base of matrix_
   MOV ES, AX       ;segment for row
   MOV DI, DX       ;mov offset to DI but save value in DX
   MOV CX, WORD ES:[DI]    ;get number of structures in row

   MOV BX, row      ;get row
   SHL BX, 1
   SHL BX, 1        ;multiply row by 4 to get offset to forcing
   MOV SI, forc         ;add base of forcing vector
   FLD DWORD [SI+BX]    ;load forcing value for row
   MOV SI, xnpl         ;get base of xnpl
   FSUB DWORD [SI+BX]   ;sub xnpl summation term

   MOV SI, xn       ;set SI to start of xn
                  ;DI has start of row, CX has # of terms in row
   CLD              ;clear direction flag
   JMP endpass1         ;skip pass1 for diagonal term
pass1:              ;start loop for second summation term
      ADD DI, 6         ;get next structure in matrix row
      FLD DWORD ES:[DI+2] ;fpush matrix value
      MOV BX, WORD ES:[DI]     ;get j index of matrix value
      SHL BX, 1
      SHL BX, 1          ;multiply j by 4 for offset to xn
      FMUL DWORD [SI+BX]  ;Aij * Xj on fstack top
      FSUB               ;subtract value from summation
endpass1:
   LOOP pass1       ;go to start of loop
                  ;summation is on stack top
   MOV DI, DX       ;get offset to start of row
   MOV CX, ES:[DI]      ;CX has # of terms in row
   FMUL QWORD weight_  ;multiply weighting factor
   FDIV DWORD ES:[DI+2]     ;divide by diagonal element

   FLD1             ;now get (1 - weight) * xn(i)
   FSUB QWORD weight_  ;(1 - weight) on stack top
   MOV BX, row      ;retrieve offset for row
   SHL BX, 1
   SHL BX, 1        ;multiply offset by four
   FMUL DWORD [SI+BX]  ;now have (1 - weight) * xn(row)
   FADD             ;now new value for xnp1 is on stack top
   MOV SI, xnp1         ;get base of xnp1
   FST DWORD [SI+BX]   ;store value in xnp1(I)
              ;xnp1(i) still on fstack top
              ;DI has start of row, CX has # of terms in row
   CLD          ;clear direction flag
   JMP endpass2         ;jump to end of loop to dec CX once
pass2:                  ;start of pass2
      ADD DI, 6         ;get next structure in row
      FLD DWORD ES:[DI+2] ;fpush matrix value
      FMUL ST, ST(1)       ;multiply by xnp1(i)
      MOV BX, WORD ES:[DI]     ;get j index of matrix value
      SHL BX, 1
      SHL BX, 1         ;multiply j by 4 for offset to xn
      FADD DWORD [SI+BX]  ;add to term in xnp1(j)
      FSTP DWORD [SI+BX]  ;store summation in xnp1(j)
endpass2:
      LOOP pass2        ;loop to end of row
   FSTP ST(0)
   JMP startit
endit:
   MOV SP, BP       ;restore SP
   POP BP           ;restore BP
   RET           ;return to caller

;==================================================================
;  ***************************************************************
;  *   COPY DOUBLE PRECISION ARRAY TO SINGLE PRECISION ARRAY   *
;  *   dfcopy(double, single, dimension)          *
;  ***************************************************************

PUBLIC dfcopy_          ;copies a double precision vector
dfcopy_:             ;to a single precision vector
   PUSH BP           ;option base 0
   MOV BP, SP
   MOV CX, WORD[BP + 8]     ;dimension of vectors
   MOV DI, WORD[BP + 4]     ;start of double precision vector
   MOV SI, WORD[BP + 6]     ;start of single precision vector
copy1:
   FLD QWORD[DI]
   FSTP DWORD[SI]
   ADD DI, 8
   ADD SI, 4
   LOOP copy1
   FWAIT
   POP BP
   RET

;==============================================================
;  *************************************************
; *   ROUTINE TO TEST CONVERGENCE OF      *
; *   GAUSS - SIEDEL ITTERATION        *
; *   (int) converge(xi, xi+1, dim, tolerance)*
; *   xi points to double pre. vector     *
; *   xi-1 points to single pre. vector   *
; *   dim is unsigned              *
; *   tolerance is double precision       *
; *   returns 1 if converged within tolerance *
; *   returns 0 if not             *
; *********************************************

PUBLIC converge_

converge_:
   PUSH BP           ;save base pointer
   MOV BP, SP        ;set base pointer for locals
   ADD SP, 2         ;make room for status word
   MOV BX, 0         ;set i to 0
   MOV DI, WORD[BP+4]  ;get start of xi
   MOV SI, WORD[BP+6]  ;get start of xi+1
   FLD QWORD [BP+10]   ;fpush tolerance

startcon:            ;start of loop
   INC BX            ;increment i
   ADD SI, 4         ;ready for next xi+1 - start at x[1]
   ADD DI, 4         ;and ready for next xi

   CMP BX, WORD[BP+8]  ;compare i with dim
   JG endloop         ;if loop is complete, go to end

   FLD DWORD [DI]        ;fpush xi[i]
   FLD DWORD [SI]        ;fpush xi+1[i]

   CALL ftest_       ;see if xi+1[i] = 0.0
   JNE change        ;if not zero then test against tolerance
   FSTP ST(0)        ;if zero, pop it off of the stack
   CALL ftest_       ;test xi[i] against 0.0
   FSTP ST(0)        ;pop xi-1[i] off of fstack
   JNE no_conv       ;if it is not zero, not converged
   JMP startcon          ;else if both are equal then goto start

change:                  ;come here if xi != 0
   FSUB ST(1), ST        ;ST = xi+1[i], ST(1) = xi+1[i] - xi[i]
   FDIVP ST(1), ST       ;ST = (xi-i[i] - xi[i])/xi[i]
   FABS              ;get absolute value
   FCOMP ST(1)       ;compare st to st(1)
   FSTSW WORD [BP-2]     ;store status word
   FWAIT
   MOV AH, BYTE [BP-1]
   SAHF              ;load flags
   JB startcon       ;if value changes by less than tolerance
                 ;then keep testing until end of array

no_conv:             ;else itteration has not converged.
   MOV AX, 0         ;set convergence indicator off
```

```
    JMP endconv        ;goto end of routine

endloop:               ;end of loop if converged
    MOV AX, 1          ;set convergence indicator on
endconv:
    FSTP ST(0)         ;pop tolerance off of fstack
    FWAIT
    MOV SP, BP         ;restore stack pointer
    POP BP             ;restore base pointer
    RET           ;return to calling routine

ftest_:
    FTST               ;compare top of stack with zero
    FSTSW WORD [BP-2]   ;store status word
    FWAIT
    MOV AH, BYTE [BP-1]
    SAHF               ;load flags
    RET

;═══════════════════════════════════════════════════════════════════
;  *************************************************
;  *   ROUTINE TO GET MAX CHANGE IN ITTERATION *
;  *   OF GAUSS - SIEDEL ITTERATION            *
;  *   (unsigned) conmax(xi, xi+1, dim)        *
;  *   xi points to single pre. vector         *
;  *   xi-1 points to single pre. vector       *
;  *   dim is unsigned                *
;  *   tolerance is double precision           *
;  *   returns 1 if converged within tolerance *
;  *   returns 0 if not               *
;  *************************************************

PUBLIC conmax_

conmax_:
    PUSH BP            ;save base pointer
    MOV BP, SP         ;set base pointer for locals
    SUB SP, 12         ;make room for status word and max change
    MOV BX, 0          ;set i to 0
    MOV CX, 0          ;set location of max change
    MOV DI, WORD[BP+4]  ;get start of xi
    MOV SI, WORD[BP+6]  ;get start of xi+1
    FLDZ               ;load zero
    FLDZ               ;load dummy max for start of loop
strt:                  ;start of loop
    FSTP ST(0)         ;pop old max off of stack
    INC BX             ;increment i

    CMP BX, WORD[BP+8]  ;compare i with dim
    JG endlp           ;if loop is complete, go to end

    ADD SI, 4          ;ready for next xi+1 - start at x[1]
    ADD DI, 4          ;and ready for next xi
    FLD DWORD [DI]         ;fpush xi[i]
    FLD DWORD [SI]         ;fpush xi+1[i]
```

```
   FTST
   FSTSW WORD [BP-12]
   FWAIT
   MOV AH, BYTE [BP-11]
   SAHF
   JNE chng        ;if not zero then test against tolerance
   FSTP ST(0)      ;else pop one off of fstack
   JMP strt        ;go to the beginning of the loop

chng:              ;come here if xi != 0
   FSUB ST(1), ST      ;ST = xi+1[i], ST(1) = xi[i] - xi+1[i]
   FDIVP ST(1), ST     ;ST = (xi-i[i] - xi[i])/xi[i]
   FABS            ;get absolute value
   FCOM ST(1)      ;compare st to st(1)
   FSTSW WORD [BP-2]   ;store status word
   FWAIT
   MOV AH, BYTE [BP-1]
   SAHF            ;load flags
   JBE strt        ;if value changes by less than tolerance
               ;then keep testing until end of array
   MOV CX, BX      ;else set pointer to max
   FXCH ST(1)      ;exchange new max for old
   JMP strt        ;loop until end of arrays

endlp:             ;end of loop if converged
   MOV AX, CX      ;put pointer into AX
   FSTP ST(0)          ;pop max off of fstack
   MOV SP, BP
   POP BP
   RET
```

```
/*

FILE NAME : PLANE.C    */
/*         ********************************************
           *  ROUTINES TO GENERATE ELEMENT STIFFNESS  *
           *  MATRIX FOR EIGHT NODE QUADRALATERAL     *
           *  ISOPARAMETRIC ELEMENT                   *
           ********************************************/
#include "c:math.h"

int nint, nintt;

double ntc[9][2] = {
   {-0.577350269189626, 1.000000000000000 },   /* two point quadrature */
   { 0.577350269189626, 1.000000000000000 },

   {-0.774596669241483, 0.555555555555556 },   /* three point quadrature   */
   { 0.000000000000000, 0.888888888888889 },
   { 0.774596669241483, 0.555555555555556 },

   { 0.861136311594053, 0.347854845137454 },   /* four point quadraure */
   { 0.339981043584856, 0.652145154862546 },
   {-0.339981043584856, 0.652145154862546 },
   {-0.861136311594053, 0.347854845137454 }

};
unsigned _showsp();

/*
       ********************************************************
       *   ISOPARAMETRIC QUADRILATERAL STRESS -                *
       *   STRAIN STIFFNESS MATRIX GENERATOR                   *
       *  isoparm(coord, thick, elnode, mat_stiff, el_matrix)  *
       ********************************************************/

isoparm(coord, thick, elnode, CC, element)
                              /* routine to generate element */
                              /* stiffness matrix */
int elnode[8];                /* points to rows of x for nodes */
float coord[][2];             /* x[i][0] - x coord node i */
                              /* x[i][1] - y coord node i */
float thick[];                /* thick[i] - thickness node i */
double CC[3][3];              /* material stiffness matrix */
double *element;              /* element stiffness matrix */
                              /* indexed for u1,v1,u2,v2,...  */
{

double Jacob[2][2];
double const, temp, *B, *B1, jacobinv(), N();
int i, j, k, l, m, posdef();
double xcc, ycc;              /* ksi, eta coordinates */
int ia, ja;
extern int nint, nintt;
extern double ntc[][2];
```

```
int posdef();
double B1[48];                          /* intermediate matrix  */
double B[16];                           /* strain matrix        */

   for (i = 0; i < 136; i++)            /* empty element */
           *(element + i) = 0.0;        /* stiffness matrix */

   for (ia = 0; ia < nint; ia++){          /* for each of the x absciscas */
       for (ja = 0; ja < nint; ja++){      /* for each of the j absciscas */

           xcc = ntc[ia + nintt][0];           /* coordinates */
           ycc = ntc[ja + nintt][0];

/* form Jacobian matrix */
           formjac(Jacob, elnode, coord, xcc, ycc);


/* form constant [N][t]|j|WiWj */
           const = 0.0;
           for (i = 0; i < 8; i++)     /* [N](t)   */
               const += N(i, 0, xcc, ycc) * thick[elnode[i]]
                        * ntc[ia + nintt][1] * ntc[ja + nintt][1];

           const *= jacobinv(Jacob);
               /* jacobinv inverts Jacob and returns determinant */

/* form [B] matrix in condensed form */
           formB(B, Jacob, xcc, ycc);

/* form [B1] = [B]tr[CC] */
           for (i = 0; i < 8; i++){        /* for each pair of rows */
               k = 6*i;                    /* start of row pair    */
               for(j = 0; j < 3; j++){     /* for each column */
/* in 1st row */   *(B1 + k+j) = *(B +i) * CC[0][j] + *(B + i+8) * CC[2][j];
/* in 2nd row */   *(B1 + k+j+3) = *(B + i+8) * CC[1][j] + *(B + i) * CC[2][j
               }
           }

/* now do [element] += const * [B1] * [B]  */
/* [element] is symmetric - only do lower triangular part */
           m = 0;                              /* initialize m */
           for (i = 0; i < 16; i++){   /* for each  row of [element] */
               k = 3*i;                /* start of row of [B1] */
               for (j = 0; j <= i; j++){   /* in each column   */
                   l = j/2;            /* l is column of [B]   */
                                       /* j is column of [element]*/
                   if (j%2)            /* if odd column */
                       *(element + m) += const * (*(B1 + k+1) * *(B+l+8) + *(
                   else                    /* else if even column */
                       *(element + m) += const * (*(B1 + k) * *(B+l) + *(B1 +

                   m++;                /* pointer to next [element] */
               }
           }
       }
```

```
    }
    if(posdef(element)){
        scr_rowcol(0,0);
        printf("Bad element");
        exitl(0);
    }
    return;

}
/*******************************************************************/

double jacobinv(jac)        /* inverts jacobian and returns det[jac] */
double jac[2][2];
{

double jacinv[2][2];
int i, j;
double det;

    det = jac[0][0] * jac[1][1] - jac[1][0] * jac[0][1];
    if (det == 0.0){
        printf("No Jacobian");
        getchar();
        exitl(0);
    }

    for (i = 0; i <2; i++){     /* copy jac to jacinv */
        for (j = 0; j <2; j++){
            jacinv[i][j] = jac[i][j];
        }
    }

    jac[0][0] = jacinv[1][1] / det;
    jac[1][1] = jacinv[0][0] / det;
    jac[0][1] = -jacinv[0][1] / det;
    jac[1][0] = -jacinv[1][0] / det;

    return (det);
}

/*******************************************************/
formjac(Jacob, elnode, coord, xx, yy)
double Jacob[2][2];         /* jacobian matrix */
unsigned elnode[8];             /* points to rows of x for each node */
float coord[][2];       /* coordinate matrix */
double xx, yy;          /* ksi, eta coordinates where Jacob formed */
{
int i, j, k;
double const;
double N();

    for (i = 0; i < 2; i++){        /* calculate in row major */
        Jacob[i][0] = Jacob[i][1] = 0.0;    /* zero row out */
        for (k = 0; k < 8; k++){    /* sum terms in row */
            const = N(k, i+1, xx, yy);  /* derv. of N wrt xory */
            for (j = 0; j < 2; j++){     /* move across row */
                Jacob[i][j] +=  const * coord[elnode[k]][j];
            }
        }
    }                       /* Jacobian has been formed */
    return;
}

/************************************************************/
formB(B, Jacob, xx, yy)
double B[2][8];                   /* pointer to start of B matrix */
double Jacob[2][2];           /* jacobian matrix has been inverted*/
double xx, yy;            /* eta, ksi coordinates where B evaluated*/
{
int i, j, k;

    for (i = 0; i < 2; i++){                /* in each row      */
        for (j = 0; j < 8; j++){            /* for each column  */
            B[i][j] = 0.0;
            for ( k = 0; k < 2; k++ )        /* row-vector product */
                B[i][j] +=  Jacob[i][k] * N(j, k+1, xx, yy);    /* dN/dx,y */
        }
    }
    return;
}
/*********************************************************/
formeps(disp, elnode, coord, x, y, epsilon)
double x, y;
float coord[][2], *disp;
double epsilon[2][2];
unsigned elnode[8];
{
unsigned l, j, k;
double Jacob[2][2];
double B[2][8];

    formjac(Jacob, elnode, coord, x, y);
    jacobinv(Jacob);
    formB(B, Jacob, x, y);
            /* [B] matrix has been formed */

/*          top row  -> du/dx   dv/dx        */

    for (j = 0; j < 2; j++){     /* calculate strains at the node */
        for (k = 0; k < 2; k++){
            epsilon[j][k] = 0.0;

            for (l = 0; l < 8; l++)      /* row column product   */
                epsilon[j][k] += B[j][l] * *(disp + elnode[l]*2 + k - 1);
                            /*   dN1/dXj                   Ul      */
        }
    }
    return;
}
/****************************************************************/
```

```
location(xx, yy, coord, elnode)        /* convert parent coordinates   */
double *xx, *yy;                /* xx, yy into real coordinates */
float *coord;
unsigned elnode[8];
{
   unsigned i;
   double temp;
   double xp, yp;

   xp = *xx;
   yp = *yy;
   *xx = *yy = 0.0;

   for (i = 0; i < 8; i++){
       *xx += (temp = N(i, 0, xp, yp)) * *(coord + 2 * elnode[i]);
       *yy += temp * *(coord + 2 * elnode[i] + 1);
   }
   return;
}

/*=====================================================================
          ****************************************
          *   TEST IF ELEMENT MATRIX POS DEF   *
          ***************************************/

int posdef(element)
double *element;
{
   int i, j;
   double *m;

   m = element;
   for (i = 0; i < 16; i++){
       if (*((m = m + i) + i) <= 0.0)
           return(1);
   }

   return(0);
}
```

```
/*

FILE : SEREND.C         */
/*══════════════════════════════════════════════════════════════
          ****************************************************
          *   ROUTINE TO GENERATE SERENDIPITY                *
          *   INTERPOLATION FUNCTIONS.                       *
          *   (double)interp_fn = N(ser_n, der_n, x, y)      *
          ****************************************************/

double N(n, d, xx, yy) /* returns value of n-th serendipity function */
                   /* function at point x, y (ksi, eta)    */
int n, d;          /* d - derivative of Nn */
                   /* d = 0 - serendipity function */
                   /* d = 1 - derivative w.r.t. x */
                   /* d = 2 - derivative w.r.t. y */
double xx, yy;     /* coordinates in ksi, eta - between -1 & 1 */

{
static int a[4][2] ={           /* coefficients for evaluation */
      {  1,  1 },               /* coordinates of corner nodes */
      {  1, -1 },               /* in ksi, eta coordinates  */
      { -1, -1 },
      { -1,  1 }
   };
double val;

   if (n < -1 || n > 7 || d < 0 || d > 2){
       printf("\nError in serendipity function\n");
       exit1(0);
   }
   xx *= a[n/2][0];
   yy *= a[n/2][1];


   if (n == -1){                /* if null serendipity function */
       return(0.0);

   } else if (!(n % 2)){            /* for corner nodes    */
       switch (d) {
           case (0):            /* return value of serendipity fn */
               val = 0.25 * (1.0 + xx) * (1.0 + yy) * (xx + yy - 1.0);
               /*               (1 + x)     (1 + y)        (x + y - 1) */
               break;

           case (1):            /* return value for derivative w.r.t. x */
               val = 0.25 * a[n/2][0] * (1.0 + yy) * (yy + 2.0 * xx);
                   /*                   (1 + y)       (y + 2 * x)       */
               break;

           case (2):            /* return value for derivative w.r.t. y */
               val = 0.25 * a[n/2][1] * (1.0 + xx) * (xx + 2.0 * yy);
               /*                       (1 + x)       (x + 2 * y)       */
           break;
```

```
        }

    } else if ((n == 1) || (n == 5)){       /* middle nodes on x axis    */
        switch (d) {
            case (0):              /* return value of serendipity fn */
                val = 0.5 * (1.0 + xx) * (1.0 - yy * yy);
                break;

            case (1):              /* derivative w.r.t x */
                val = 0.5 * a[n / 2][0] * (1.0 - yy * yy);
                break;

            case (2):              /* derivative w.r.t y */
                val = -yy * a[n / 2][1] * (1.0 + xx);
                break;
        }

    } else {             /* middle nodes on y; n = 3 or 7    */
        switch (d) {
            case (0):              /* return value of serendipity fn */
                val = 0.5 * (1.0 - xx * xx) * (1.0 + yy);
                break;

            case (1):              /* derivative w.r.t x */
                val = -xx * a[n / 2][0] * (1.0 + yy);
                break;

            case (2):              /* derivative w.r.t y */
                val = 0.5 * a[n / 2][1] * (1.0 - xx * xx);
                break;
        }
    }
    return(val);
}
```

```
/*

FILE : FGRAPH1.C       */
/*==========================================================================
      ****************************************************
      *   GRAPHICS ROUTINES FOR FINITE ELEMENT PROGRAM   *
      ****************************************************/

#include "graphics.h"
#include "math.h"
#define crd(i, j)  cord[i][j]  /* macro for matrix addressing  */

double xmin, ymin, deltax;     /* min and span values to be plotted */
double xming, yming, deltaxg;  /* global min and span values    */
double voverh;                 /* vertical over horizontal ratio   */
unsigned vdots, hdots;         /* vertical and horizontal res  */
unsigned vidseg;               /* video segment start          */
int folor, bolor, palette, bright; /* color parameters */
unsigned pbuff, pntf, pntfg;       /* printer buffer and flags     */
/*=========================================================================
      ************************************************
      *   INITIALIZE SYSTEM FOR GRAPHICS ROUTINES    *
      ************************************************/

initgraph(cord, dim)
   unsigned dim;
   float cord[][2];
{
   folor = 2;                  /* initialize variables */
   bolor = 0;
   bright = 16;
   voverh = 3.0/4.0;                     /* set display ratio    */
   vdots = 200;
   hdots = 320;
   vidseg = 0xb800;                      /* set video segment    */

   window(voverh, cord, dim);            /* set window parameters   */
}

window(voverh, cord, dim)
double voverh;
float cord[][2];
unsigned dim;
{
   unsigned i;
   double temp;
   double deltay, xmax, ymax;

   xmax = xmin = crd(1,0);
   ymax = ymin = crd(1,1);
   for (i = 1; i <= dim; i++){          /* find max, min coords */
       xmax = (xmax < crd(i, 0)) ? crd(i, 0) : xmax;
       xmin = (xmin > crd(i, 0)) ? crd(i, 0) : xmin;
       ymax = (ymax < crd(i, 1)) ? crd(i, 1) : ymax;
```

```
        ymin = (ymin > crd(i, 1)) ? crd(i, 1) : ymin;
    }
    deltax = (xmax - xmin) * 1.00001;                 /* x span   */
    deltay = (ymax - ymin) * 1.00001;                 /* y span   */

    if (deltay > voverh * deltax){             /* if still too high    */
        xmin -= 0.5 * (deltay/voverh - deltax);     /* adjust xmin  */
        deltax = deltay / voverh;              /* adjust deltax        */
    } else {
        ymin -= 0.5 * (voverh * deltax - deltay);   /* adjust ymin  */
        deltay = voverh * deltax;
    }

    xming = xmin;
    yming = ymin;
    deltaxg = deltax;
    scr_setup();
    return;
}
/*═════════════════════════════════════════════════════════════════
       *****************************************************
       *   CONVERT REAL COORDINATES TO SCREEN COORDINATES  *
       *****************************************************/
felplot(x, y, c)
double x, y;
int c;
{
    int xpoint, ypoint;
    extern double xmin, ymin, deltax;

#define xmax (xmin+deltax)
#define ymax (ymin+voverh*deltax)
    if (x < xmin || y < ymin || x > xmax || y > ymax)
        return;                  /* clip */

    xpoint = (hdots-1.0) * (x - xmin)/ deltax + 0.5;
    ypoint = (vdots-1.0) * (1.0 - (y - ymin)/(deltax * voverh)) + 0.5;
    if (!pntf)                               /* if write to screen   */
        scr_wdot(xpoint, ypoint, c);
    else                                     /* else if write to printer */
        prn_wdot(xpoint, ypoint);

    return;
}
/*═════════════════════════════════════════════════════════════════
                 *********************************************
                 *   DRAW AN ELEMENT ON THE GRAPHICS SCREEN  *
                 *   NOTE: THIS IS USED BY FINEL             *
                 *********************************************/
eldraw(n, coord, elnode)
float coord[][2];
unsigned n;                             /* element number   */
unsigned elnode[][8];

{
   int i, j;
   static double x[3][2];

   if (folor)
       for (i = 0; i < 8; i += 2){
           for (j = 0; j < 3; j++){
               x[j][0] = coord[elnode[n][(i+j)%8]][0];
               x[j][1] = coord[elnode[n][(i+j)%8]][1];
           }
           isoline(x);
       }

   for (i = 0; i < 8; i++)              /* plot element nodes        */
       felplot(coord[elnode[n][i]][0], coord[elnode[n][i]][1], (folor-1)&3);
}
/*=====================================================================
           ******************************************************
           *   DRAW ISOPARAMETRIC LINE THROUGH THREE POINTS     *
           ******************************************************/
isoline(x)
   double x[][2];
{
   int i;
   double xx, yy, temp;
   double dx, dy;          /* two different steps, use minimum */
   double eta, deta;       /* parent line coordinate           */
   double a1, b1, c1, a2, b2, c2;
   extern double deltax;
   extern int folor;

   a1 = 0.5 * (x[0][0] + x[2][0] - 2 * x[1][0]);
   a2 = 0.5 * (x[0][1] + x[2][1] - 2 * x[1][1]);
   b1 = 0.5 * (x[2][0] - x[0][0]);
   b2 = 0.5 * (x[2][1] - x[0][1]);
   c1 = x[1][0];  c2 = x[1][1];

   eta = -1.0;
   while (eta <= 1.0){
       xx = (a1 * eta + b1) * eta + c1;
       yy = (a2 * eta + b2) * eta + c2;
       felplot(xx, yy, folor);
       dx = fabs(2.0 * a1 * eta + b1) * hdots;
       dy = fabs(2.0 * a2 * eta + b2) * vdots / voverh;
       deta = (dx > dy) ? dx : dy;
       eta += (deta) ? deltax / deta : 2.0;
   }
   return;
}
/*====================================================================
               **********************************************
               *   GENERATE DERIVATIVE OF 1 DIMENSIONAL     *
               *       INTERPOLATION FUNCTION               *
               **********************************************/
```

```
double N1d(i, eta)
   int i;
   double eta;
{
   double val;
   switch (i) {
       case 0:
           val = (0.5 * (1.0 + 2.0 * eta));
           break;
       case 1:
           val = (-2.0 * eta);
           break;
       case 2:
           val = (0.5 * (2.0 * eta - 1.0));
           break;
       default:
           printf("\nError in 1-D interpolation");
           exitl(0);
   }
   return(val);
}

/*=====================================================================
                            SELECT SCREEN                          */
scrn(n)                        /* configure the screen     */
int n;                      /* mode = 0 for A/N, 1 for graphics */
{

   switch (n){
   case 0:                     /* if A/N screen         */
       screen(0);              /* set A/N screen        */
       width(80);
       scr_color(0, 1);        /* set border color      */
       break;
   case 1:
       screen(1);                          /* set 320 by 200        */
       scr_color(palette, bolor|bright);   /* select color and palette */
       break;
   }
   return;
}
```

```
; File: Fext.a

;===============================================================
;     ROUTINES TO HANDLE DATA IN 6 BYTE MATRIX STRUCTURES
;     (unsigned) valfu(n) return the unsigned value
;     (double) valfd(n)   return double value
;     storefu(n, i)       store unsigned value
;     storefd(n, x)       store double value
;     addfd(n, x)     add double value
DSEG
PUBLIC matrix_
CSEG
PUBLIC valfu_
valfu_:
   MOV SI, SP
   MOV AX, WORD [SI+2]      ;get n
   MOV CX, 6            ;multiply n by 6/16
   MUL CX
   MOV CX, 16
   DIV CX
   ADD AX, matrix_          ;add matrix to get segment
   MOV DI, DX           ;put offset to si
   MOV ES, AX           ;get segment in ES
   MOV AX, WORD ES:[DI+2]
   RET

PUBLIC storefu_
storefu_:
   MOV SI, SP
   MOV AX, WORD [SI+2]      ;get n
   MOV CX, 6            ;multiply n by 6/16
   MUL CX
   MOV CX, 16
   DIV CX
   ADD AX, matrix_          ;add matrix to get segment
   MOV DI, DX           ;put offset to si
   MOV ES, AX           ;get segment in ES
   MOV AX, WORD [SI+4]      ;get unsigned value
   MOV WORD ES:[DI+2], AX       ;store in matrix element
   RET

PUBLIC valfd_
valfd_:
   MOV SI, SP
   MOV AX, WORD [SI+2]      ;get n
   MOV CX, 6            ;multiply n by 6/16
   MUL CX
   MOV CX, 16
   DIV CX
   ADD AX, matrix_          ;add matrix to get segment
   MOV DI, DX           ;put offset to si
   MOV ES, AX           ;get segment in ES
```

```
    FLD DWORD ES:[DI+4]
    RET

PUBLIC storefd_                  ;storefd(n, double)
storefd_:                   ;store double value in matrix element
    MOV SI, SP
    MOV AX, WORD [SI+2]        ;get n
    MOV CX, 6               ;multiply n by 6/16
    MUL CX
    MOV CX, 16
    DIV CX
    ADD AX, matrix_            ;add matrix to get segment
    MOV DI, DX              ;put offset to si
    MOV ES, AX              ;get segment in ES
    FLD QWORD [SI+4]           ;fpush double value
    FSTP DWORD ES:[DI+4]          ;store in matrix
    RET

PUBLIC addfd_                    ;addfd(n, double)
addfd_:                          ;add double value to matrix element
    MOV SI, SP
    MOV AX, WORD [SI+2]        ;get n
    MOV CX, 6               ;multiply n by 6/16
    MUL CX
    MOV CX, 16
    DIV CX
    ADD AX, matrix_            ;add matrix to get segment
    MOV DI, DX              ;put offset to si
    MOV ES, AX              ;get segment in ES
    FLD QWORD [SI+4]           ;fpush double value
    FADD DWORD ES:[DI+4]          ;add value in matrix
    FSTP DWORD ES:[DI+4]          ;store in matrix
    RET
```